甘利俊一　麻生英樹　伊庭幸人　編

確率と情報の科学

高速文字列解析の世界

データ圧縮・全文検索・テキストマイニング

岡野原大輔

岩波書店

まえがき

文字列データは広い分野でみられる一般的なデータである．文書，ウェブページ，生物のゲノム配列，センサデータ，行動履歴など，ありとあらゆる情報が文字列で表現されている．

こうした文字列データは近年，急速に増加している．個人がソーシャル空間で日常の生活ログを発信し，センサデバイスからは多種多様な情報が休むことなく集められ続け，ゲノムシーケンサはこれまでとは比較にならない速度でゲノム配列を読み取ることが可能となっている．

人が一生に読む量の数万倍から数億倍という膨大な量の文字列を精緻に解析するために，文字列解析技術は大きな発展を遂げてきている．必要は発明の母というが，古くからある文字列解析も21世紀に入ってから日進月歩で技術革新が起こり，大規模なデータを非常に効率よく扱えるようになった．

文字列解析は身近な技術であり，生活の中にありふれている．例えば，ウェブ上で検索エンジンを利用した場合，ユーザーが入力したクエリ文字列と合致するウェブページを膨大な量の候補から瞬時に探し出すために，さまざまな文字列解析技術が利用されている．

文字列サイズが小さい場合は，どんな手法を用いて解析を行ったとしても，計算コストが問題にはならない．しかし，今日扱おうとしている文字列データは非常に大きく，単純な手法はすぐに破綻する．この問題は計算リソースを増やせば解決する問題でもない．分析対象のデータ量に線形に比例して計算量が増えるのであれば，計算リソースの追加で解決できるが，単純に見える文字列解析の多くは線形よりも大きく計算量が増え，うまく解かないと計算量は爆発してしまう．大規模なデータを扱うときにこそ，優れたアルゴリズムやデータ構造を選び抜き利用しなければならない．

文字列解析の分野は広く，進展も著しいため，本書ですべての情報はカバーできない．本書では文字列解析という問題の特徴，面白さをとらえており，普遍的である話を取り込むことを心がけた．特に，Burrows Wheeler 変換，簡

潔データ構造，ウェーブレット木という3つの技術に注目し，解説を試みた．これら3つの技術はここ十年における文字列解析の中で最も大きな進歩であり，今後も重要な技術と考えられる．そして，これらの技術は文字列解析以外にもグラフ情報や，2次元グリッド情報などに対して非常に強力なツールとなる．

文字列解析を実際に使ってみると，理論と実践の間にギャップがみられやすい．計算量のオーダーが少ないからといって，必ずしも優れている手法とは限らない．単純で汎用的であり，最悪時の挙動が抑えられつつ，実際に扱うデータに対して効率的であることが求められる．本書は，理論だけを追うのではなく実用的な側面も重視し，データや現在の計算機環境のトレンドを考慮し，シンプルで他と組み合わせやすい手法に注目して書いた．

本書を読むことによって，文字列解析の世界に少しでも興味を持っていただければ幸いである．

この本を書くにあたって，編者の麻生英樹先生，伊庭幸人先生から構成・内容についてアドバイスをいただき，また定兼邦彦先生，大野健太さんにも何度も読んでいただき，誤記・不正確・不明瞭な点のご指摘をいただきました．この場を借りて深く感謝を申し上げたい．

2012年12月

著　　者

目　次

装丁　蛯名優子

1

文字列解析の今

本章では，世の中にあふれる文字列データの例をいくつか紹介し，現在の計算機環境をふまえたうえで，どのような文字列解析が求められているかを解説する．さらに，重要な文字列解析のための道具を紹介し，本書全体の構成を説明する．

1.1 世の中の文字列

文字列データは世の中にあふれており，多様な情報が文字列で表されている．文字列という基本的なデータをどのように解析するかは，コンピュータが誕生する以前から理論，実践ともに研究され続けてきている．2000年以降，インターネットの普及と，ゲノム解析の急速な進展，さらにセンサデータや行動履歴データなども文字列データとして記録されるようになり，文字列データは爆発的に増加した．そして，こうしたデータを解析し，活用したいという要求の高まりに応じて，文字列解析の技術は急速に進展している．以降では，世の中で実際にみられる文字列データの例をいくつか見てみよう．

1.1.1 言語情報

文字列データの代表例は，人が読み書きに利用している言語情報であろう．言語情報は，情報を記録し，コミュニケーションをとるための記号列であり，数千年前から，さまざまな事象や知識，そして人の考えが記録されてきた．言語情報は本やメール，法律，技術論文，契約書など形式や内容は多彩である．

また，インターネットの普及により人々は有史以来，最も盛んに言語情報を介してコミュニーケーションをとり，日々の感情や体験をウェブ上で共有するようになっている．例えば，ソーシャルメディア上には数億人の発言が日々飛び交い，その数は秒間数千から数万にもなる．今後は音声認識がユーザーインターフェースとして普及するとともに，音声データから得られる言語情報も急激に増加することが予想されている．

こうした言語情報の解析を通して，人々の行動や考えを理解し，埋もれていた重要な情報を発見することができる．

1.1.2 生物の遺伝情報

この半世紀で，生命の遺伝情報はDNAやRNAと呼ばれる核酸からなる文字列，いわゆるゲノム配列によって記録されていることが明らかになってきた．

これらの遺伝情報や，その制御の仕組みなどの解明を通して，生命科学は大きな進展を遂げつつある．そして，解析機器の急速な改良にも後押しされて世界中のさまざまな実験結果が集められ，膨大な量の遺伝情報の解析が進められている．

ゲノム解析は生命科学のみならず医療や農業など，広い分野で大きな役割を果たしていく．今や，読み取られたゲノム配列をもとに，どのような遺伝子が発現しているのか，個体間の差がどこにあるのかなどを網羅的に調べることが可能となっている．今後は，例えば個人ごとに適切な医療行為を選んで提供できるオーダーメイド医療が実現できるだろう．こうした解析を支えるために，遺伝情報を表す膨大な文字列データを，効率的に処理する方法が求められている．

1.1.3 ログ情報，行動履歴情報

コンピュータが何をしたか，誰といつアクセスしたかといったログ情報は，文字列という形で記録される．こうしたログ情報は決められたフォーマットで書かれた場合もあれば，自由な形式で書かれている場合もある．行動履歴についても，例えばウェブ上での行動であれば，誰がどのウェブサイトをいつ訪問したかが記録され，それらは文字列情報として表せる．また，センサデバイスが普及するにつれ，集められた情報の解析が重要となる．こうしたセンサデバイスから集められるのは温度や振動，画像といった連続的な数値情報であるが，それらから特徴情報を抽出し，離散的な記号列，つまり文字列情報として扱うことができる．例えば，前の日と比べて気温が上昇した，下降した，変わらなかったという情報をそれぞれ U, D, S という文字を使って，気温の変化の時系列情報を文字列として表すことができる．

どのような情報であっても，いったん文字列データに変換できれば文字列解析のさまざまな技術を利用でき，膨大な量の解析を効率的に行うことができる．今後センサデバイスから集められるデータが爆発的に増加するにつれ，これらのデータを高速かつ緻密に解析するために，文字列解析はさらに重要となる．

すなわち今後，ログ，行動履歴，センサデバイスが生み出す情報は，量，生

成速度ともに非常に大きく，高度な解析技術が必要とされる．

1.2　現在の計算機環境

実際に利用する場面でうまく動くかを考えるには，現在の計算機環境にどのような特徴があるのかを理解しておかなければならない．

2012 年までの計算機環境は，一貫して演算速度は速くなる一方で，記憶装置へのアクセス速度は一定のままというトレンドがみられている．一般に記憶装置は，上位の階層になるほどサイズは小さく，高速であるという特徴がある．そこで，高速な記憶装置(レジスタ，キャッシュ，メモリ)にいかにデータを格納して演算を行うかが，高速なデータ解析の実現のポイントとなる．現在では，データをコンパクトに格納することと，データを高速に解析することは同義である．

記憶装置の階層間の速度差が大きくなり，演算回数が増えたとしても，データを小さくでき，上位の高速な記憶領域で処理することができれば，全体としては高速化を達成できる．例えば，2012 年現在の標準的な計算機では，最上位である L1 キャッシュへのアクセスは約 0.5 ナノ秒かかるのに対し，下位のメインメモリへのアクセスは約 100 ナノ秒かかり，200 倍の速度差がある．また，データを 1 M バイト読み込む場合，メインメモリからは約 0.2 ミリ秒かかるのに対し，ハードディスクからは約 20 ミリ秒かかり，ここでも 100 倍の速度差がある．文字列解析においてはデータアクセスがボトルネックになりやすいので，上位の記憶階層で処理することは数百倍から数千倍の高速化につながる．また，今後はクラウド上での処理が増えていくことが考えられる．各端末で発生したデータは，クラウド上にアップロードされたうえで，クラウド上で解析が行われ，解析結果が返される．その際にはデータの通信コストが問題となり，送信するデータ量をいかに削減できるかが，高速化とコスト削減につながる．

大量のデータをいかに効率よく格納するか，そして格納したうえでいかに効率よく処理するかが，文字列解析における最重要課題となる．

1.3　文字列解析の例

文字列解析とはどのようなものか，感触をつかんでもらうために，次の問題を考えてみよう．

問題 1.1　文字列 T, U が与えられたとき，T のみに出現する部分文字列をすべて，漏れなく重複なく列挙せよ． □

この問題が実世界でみられる例を，以下にいくつか示そう．正常に動作している機器と故障している機器に設置された2つのセンサから時系列データが貯められ，それらが文字列で表現されているとする．このとき，故障している機器だけにみられる部分文字列を見つけられれば，今後センサから同じ部分文字列を観測した場合に故障を検知できるかもしれない．同じような問題は言語判定でもみられる．与えられた文字列が何語(英語，ドイツ語，イタリア語，その他)であるかの判定には，ある言語のみで出現するような単語を見つけることが重要な手がかりとなる．しかし，日本語のように分かち書きされていない言語の場合には，あらかじめ単語に区切ることが困難で，同じようには扱えない．そこで手がかりの対象を単語ではなく任意の部分文字列とし，それぞれの言語でのみ現れる部分文字列を見つけることができれば，言語判定ができるだろう．

では，この問題を考えてみよう．そのまま単純に，T, U それぞれに出現する部分文字列をすべて列挙し，それらを比較して T のみに出現しているかをチェックした場合を考えてみる．この方法では，計算量は文字列長の2乗に比例し，文字列長が長くなるにつれ計算量は爆発的に増える．本書では，扱う文字列の長さは数十億から数百億程度を想定している．この単純な方法では文字列の長さが数万を超えた程度で，必要な作業領域量が主記憶のサイズを超え，現実的な時間とリソース量で処理することはできない．

ところがこの問題は，適切なデータ構造とアルゴリズムを利用すれば，入力長の合計に比例する計算量で解くことができ，作業領域量はもとの文字列の合計長と同じ程度で実現可能である．例えば T と U それぞれの長さが百万程度であれば1秒以下，数十億であっても数分程度で漏れなく列挙できる．この

方法については 8.1.2 節で扱う．

このような，一見するとありえなさそうなことが，文字列解析技術の組み合わせで可能となる．

1.4　文字列解析のための道具

本書では，広範な文字列解析の中でもデータ圧縮，全文検索，そしてテキストマイニングを対象に解説を行う．こうした技術は，これまで違う分野で別々に論じられることが多く，研究的にも実用的にも十分な交流は行われて来なかった．

例えば，データ圧縮は情報理論や文字列アルゴリズム，全文検索は情報検索やデータベース，そしてテキストマイニングはデータマイニングや機械学習の分野で別々に論じられてきた．

こうした文字列解析の技術は密接に関係しており，ヒューリスティックスの塊であるように思える文字列解析が，実は美しい理論を適用すれば実用的かつ効率的に実現できることがわかる．そして本書では，数多くある技術の中でも Burrows Wheeler 変換，簡潔データ構造，ウェーブレット木に注目した．これらの技術について簡単に紹介しておく．

1.4.1　Burrows Wheeler 変換

本書の中で重要な役割を果たす Burrows Wheeler 変換 (BWT) は，文字列を他の文字列に可逆変換する．フーリエ変換が信号情報を周波数ごとに表現する変換であるのと同様に，BWT は部分文字列の情報ごとに文字列を表現する．BWT を用いることで，もとの文字列を効率的に格納できるだけでなく，任意の部分文字列の出現回数を高速に求めるなど，さまざまな解析を行うことができる．

BWT はもともとデータ圧縮の分野で登場したが，その後，全文検索やテキストマイニングにも強力なツールであることがわかりはじめ，研究が急速に進んでいる．そして BWT はすでに，世の中のさまざまな分野で利用されている．例えばデータ圧縮ソフトウェアである bzip2 や，ゲノム解析における，

シーケンサから読み取られたゲノム配列のアライメントツールなどである．

BWT の登場自体は 1980 年頃であったが，実用的になったのは 2000 年頃からである．この理由に，BWT の効率的な計算方法がこれまで存在しなかったことがある．現在では，テキスト長に対して線形時間で，作業領域量的にも効率的な方法が確立され，非常に巨大な文字列に対しても BWT を計算できるようになった．

1.4.2　簡潔データ構造

簡潔データ構造は，データ圧縮と索引の技術が融合したデータ構造であり，情報を圧縮して格納したまま，さまざまな操作を高速に実現できる．

従来，データ圧縮といえば，一度圧縮したら復元しなければ使えなかった．そのため，巨大なデータを圧縮した後に，そのデータの一部分だけを使いたい場合でも，すべてのデータを一度復元しなければならなかった．一方，簡潔データ構造は構造を有したデータをコンパクトに格納したまま，高速な操作を実現するデータ構造である．高速な操作をサポートするために，簡潔データ構造は“索引”と呼ばれる補助データ構造を前もって構築しておき，操作の際は索引を利用することで，データ全体を見なくても正しい結果を瞬時に返すことができる．

具体例として，木構造に対する簡潔データ構造である簡潔木を紹介する．

これまでの木構造の表現では，内部節点間を移動するために各節点ごとに，その親，最初の子，次の兄弟など，隣接する定数個の節点へのポインタを格納していた．節点数が n のとき，各ポインタのサイズは $\log_2 n$ ビットとなり，全体で必要な作業領域量は $O(n \log n)$ ビットである．例えば，節点数が 2^{64} であり，親，最初の子，次の兄弟を格納する場合は，1 節点あたり $64 \cdot 3 = 192$ ビットを格納する必要があった．それに対し，簡潔木は木を 1 節点あたり 2 ビットで格納したまま，親や子，兄弟への遷移を定数時間で行うことができる．ポインタを利用した手法と比べ，簡潔木のサイズは約 1/100 である．

簡潔データ構造は木以外にも，文字列，グラフ，関数，2 次元グリッド情報といったさまざまな情報に対して提案されており，これらは文字列データ解析のみならず，その他の問題に対しても重要なデータ構造となっている．

1.4.3　ウェーブレット木

ウェーブレット木は万能のデータ構造であり，文字列解析のみならず，時系列解析，2 次元グリッドデータ，グラフ情報など，さまざまなデータに対し多くの操作を効率的にサポートする．例えば，任意の連続する範囲内の最大値，最小値，値の出現回数，最頻値といったさまざまな値を，データの大きさや範囲の広さによらない時間で求めることができる．

ウェーブレット木の大きな特徴は，2 つの属性があるデータ集合をどちらの属性でもソートされている状態で扱うことができる点である．文字列データも，位置を x 軸の値，各文字を y 軸方向の値とした 2 次元情報だと考えることができる．T=12020 という文字列は，2 次元中の $(0,1),(1,2),(2,0),(3,2),(4,0)$ に点が存在するデータだと考えることができ，位置と値の 2 つの属性があるデータと考えたうえでウェーブレット木を利用できる．また，ウェーブレット木はデータを効率的に格納でき，もとの表現方法と同じサイズか，それより小さなサイズで保存できる．さらに，ウェーブレット木は Burrows Wheeler 変換や簡潔データ構造と組み合わせることで，全文検索や情報抽出などの文字列解析処理を効率的に実現できる．

1.5　本書の概要

本書の構成を図 1.1 に示す．

第 2 章では，本書を通じて必要な文字列や符号の基本知識，概念を紹介する．最初は読み飛ばして，後で必要に応じて参照してもらってもよい．

前半の第 3〜5 章では先ほど紹介した，文字列解析に必要な 3 つの技術を紹介する．第 3 章では，Burrows Wheeler 変換と，それに関係する接尾辞配列などについて紹介する．第 4 章では膨大な量の文字列情報を効率的に格納しつつ，高速な操作を実現できる簡潔データ構造について解説を行う．第 5 章では，さまざまな文字列操作を統一的なデータ構造で効率的に扱える，ウェーブレット木を説明する．

後半の第 6〜8 章で，これらの技術を利用した文字列解析の実践を述べる．

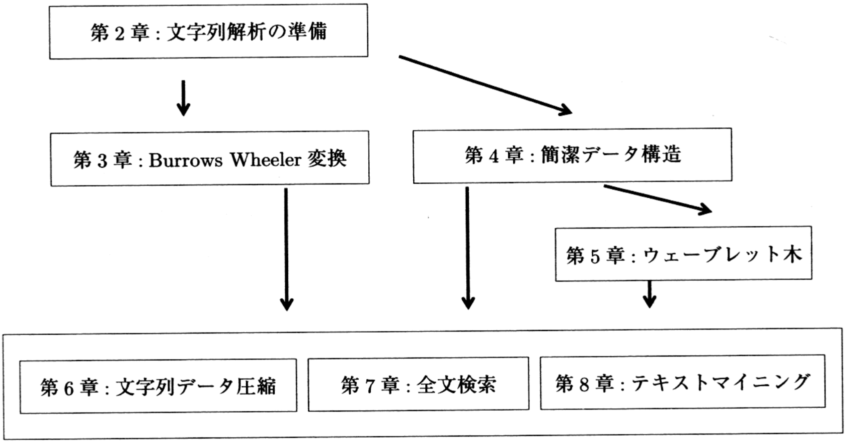


図 1.1　本書の構成

第 6 章では文字列のデータ圧縮を紹介する．データ圧縮は，データをうまく表現できる方法を求め，それに沿った形でデータを格納し，データを元の表現方法よりも少ないサイズで格納する方法である．データ圧縮は，データが規則性を持ち，冗長な場合にうまく圧縮できる．例えば，ヒトゲノム情報は 30 億個の文字からなるが，個人間の差はわずかであり，各ゲノム情報をそのまま記録するのではなく差分情報のみを記録すれば，もとと比べて遥かに小さなサイズで保存できる．現在，高速化においてデータ圧縮が重要な要素となるにつれ，データ圧縮は大規模データの効率的な解析の中心課題となっている．

第 7 章では，全文検索について紹介する．全文検索は，検索対象の文字列中に対し，クエリ文字列の出現回数や出現位置を求める問題である．全文検索は直接検索として利用するだけでなく，他の文字列解析のツールとして利用できる．多くの検索エンジンで利用される転置索引や N-gram 索引を利用した全文検索とともに，任意の部分文字列を効率的に検索できる接尾辞配列を利用した全文検索を説明する．

また，全文検索では，その索引が元テキストと同程度に大きいという課題があった．これを解決するのが圧縮全文索引である．圧縮全文索引は索引を圧縮したまま全文検索などさまざまな操作を効率的に実現するデータ構造であり，

圧縮と簡潔データ構造，そして全文検索の技術を組み合わせて作成された，文字列解析の技術の集大成と呼べる．圧縮全文索引は，もとの文字列を別に保持する必要がなく，索引のみで全文検索ができ，さらにもとの文字列を復元できるという性質がある．

第8章では，文字列解析を利用したさまざまな分析について紹介する．文字列中のあらゆる部分文字列について，各種の統計情報をいかに効率的に求めるかについて述べる．例えば，文書集合中で頻出するような部分文字列の列挙や，すべての部分文字列の統計量の計算などが，高速に実現できることを示す．

2

文字列解析の準備

本書を通して使用する文字列解析のさまざまな定義，概念を紹介し，文字列に対する操作を説明する．また文字列の複雑さを表す経験エントロピー，および文字列解析の計算量を評価するための計算モデルを解説する．最後に，文字列の表現方法として扱う符号について紹介する．

2.1　文字列の記法

はじめに文字列の記法について，いくつか紹介しておく．これらの記法は，本書を通じて利用する．

文字列を T や U といった大文字で表し，文字列 T の長さを $|T|$ で表す．各文字は c や x といった小文字で表す．特定の文字や文字列を指す場合は，ダブルクオーテーションで囲み，"*t*", "*tokyo*" のように表す．例えば，$T=$ "*abracadabra*" の場合，$|T|=11$ である．文字列 T の i 文字めから，$j\geq i$ 文字目までの部分文字列を $T[i,j]$ で表し，$T[i]=T[i,i]$ とする．また，終端を含めない部分文字列を $T[i,j)=T[i,j-1]$ とする．文字列中の文字は 0 から始まる添字で指定する．文字列を並べて表記した場合は文字列の連結を表し，TU は文字列 T の後に文字列 U を連結した文字列であるとする．

文字列 T の文字がとりうる値の集合 Σ を T のアルファベット集合と呼び，この集合のサイズを $\sigma=|\Sigma|$ とする．例えば，"*a*" から "*z*" までのアルファベットからなる文字列では $\sigma=26$ であり，DNA の場合は "A, T, C, G" の 4 種類の塩基に対応する文字から構成されるので $\sigma=4$，日本語の場合は，すべての漢字，ひらがな，カタカナからなるので σ は数万から数十万となる．本書で扱うアルファベットには順序関係が存在し，それは全順序である．すなわち異なる種類の文字なら必ず「大小」が定まるとする．特に指定がない場合は，アルファベットは $\{0,1,2,\cdots,\sigma-1\}$ の整数であるとする．また，Σ^k は各文字が Σ 中の文字からなる長さ k の文字列すべての集合とする．例えば $\Sigma=\{a,b,c\}$ のとき，$\Sigma^2=\{aa,ab,ac,ba,bb,bc,ca,cb,cc\}$ である．

文字列 $T[0,n)$ の先頭から i 番めまでの部分文字列 $T[0,i]$ を，T の接頭辞(Prefix)と呼ぶ．また，T の i 番めから最後までの部分文字列 $T[i,n)$ を T の接尾辞(Suffix)と呼び，S_i で表すとする．例えば，$T=$"*tatami*" のとき，$|T|=6$，$T[2,5]=$"*tami*"，$T[0,3)=$"*tat*"，$T[4,4]=T[4]=$"*m*" である．また，$T[0,0]=$"*t*"，$T[0,1]=$"*ta*"，$\cdots$，$T[0,5]=$"*tatami*" は T の接頭辞，$T[0,6)=$ "*tatami*", $T[1,6)=$"*atami*", $T[2,6)=$"*tami*" は T の接尾辞である．これら接尾辞と接頭辞は，文字列解析において重要な役割を果たす．文字列 T に出現

する空でない部分文字列は，部分文字列の開始位置 s と終了位置 e の選び方が，$0 \leq s \leq e < |T|$ であるので，$|T|(|T|+1)/2$ 個存在する．それに対し，文字列 T の接頭辞，接尾辞は $|T|$ 個ずつ存在する．

次に辞書式順序について説明する．2 つの文字列 T, U に対し，先頭から順に各文字列の文字を比較していったときに，初めて異なった文字の大小関係で決まる順序を辞書式順序と呼ぶ．

定義 2.1（辞書式順序：lexicographical order）

$$T <_{lex} U \Leftrightarrow T[j] < U[j] \quad \text{ただし} \quad j = \min_i \{i \in [0, |T|), |T[i] \neq U[i]\} \tag{2.1}$$

□

例えば，$aa <_{lex} abc <_{lex} abd <_{lex} ac <_{lex} ba$ である．この辞書式順序は，すべての文字列間については成り立たないことに注意する．例えば，$T=$"ab" と $U=$"abc" については，先頭 2 文字めまでは一致するが，T には 3 文字めは存在しないので比較できない．一般に T が U の接頭辞である場合，またはその逆の場合には比較できない．

最後に整数の除算，剰余を定義する．整数 $a, b \neq 0$ に対し，a/b を a を b で割った商とし，$a \bmod b$ を a を b で割ったときの余りとする．また，$a//b = (a/b) \cdot b$ とし，b の倍数で a を超えない最大の整数であるとする．例えば，$11/3=3$，$11 \bmod 3=2$，$11//3=9$ である．$a//b + (a \bmod b) = a$ という関係が成り立つ．また正整数 x に対し，$\lg x = \log_2 x$ と定義する．

2.2　文字列上の操作

次に，文字列に対してどのような操作が必要となるかを見てみる．文字列が与えられたときの代表的な操作を表 2.1 に示す．特に重要となるのが，**rank** と **select** 操作の 2 つである．例えば，$T[0, 10)=$"$abdcabcdda$" の場合，**access**$(T, 4) = T[4] =$"a"，**rank**$_{\text{"}b\text{"}}(T, 8) = 2$，**select**$_{\text{"}d\text{"}}(T, 2) = 8$ である．これらの操作を組み合わせることで，さまざまな文字列解析が可能となる．

本書ではこれらの操作を効率的に扱うデータ構造を紹介する．ビット列

表 2.1 文字列 T に対する操作

操作名	説明
access(T,i)	位置 i の文字 $T[i]$ を返す.
rank$_c(T,i)$	$T[0,i)$ 中の文字 c の出現数を返す.
select$_c(T,i)$	T 中の $(i+1)$ 番めの c の出現位置を返す.
quantile(T,s,e,r)	$T[s,e)$ 中の $(r+1)$ 番めに大きい値を返す.
topk(T,s,e,k)	$T[s,e)$ 中で出現回数が多い文字順にその頻度とともに k 個返す.
rangefreq(T,s,e,x,y)	$T[s,e)$ 中に出現する $x \leq c < y$ を満たす文字 c の合計出現数を返す.
rangelist(T,s,e,x,y)	$T[s,e)$ 中に出現する $x \leq c < y$ の各文字 c を頻度とともに列挙する.
rangemaxk(T,s,e,k)	$T[s,e)$ 中に出現する文字を大きい順にその頻度とともに k 個返す.
rangemink(T,s,e,k)	$T[s,e)$ 中に出現する文字を小さい順にその頻度とともに k 個返す.
prevvalue(T,s,e,x,y)	$T[s,e)$ 中で $x \leq c < y$ を満たす最大の c を返す.
nextvalue(T,s,e,x,y)	$T[s,e)$ 中で $x \leq c < y$ を満たす最小の c を返す.
intersect(T,s_1,e_1,s_2,e_2)	$T[s_1,e_1)$ と $T[s_2,e_2)$ の間で共通して出現する文字と頻度を返す.

$(\Sigma=\{0,1\})$ の場合に **access**, **rank**, **select** をサポートしたデータ構造を完備辞書と呼び，これは 4.2 節で扱う．また，一般のアルファベット集合の場合に表 2.1 のすべての操作をサポートしたウェーブレット木を 5.1 節で説明する.

2.3 経験エントロピー

次に，文字列の単純さ，そして圧縮しやすさを示す経験エントロピーを説明する．文字列 T の 0 次経験エントロピー $H_0(T)$ を次のように定義する.

定義 2.2(0 次経験エントロピー)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \lg \frac{n}{n_c} \tag{2.2}$$

□

ただし，n は T の文字列長であり，n_c は文字列 T 中に文字 c が出現した回数である.

この経験エントロピーは，シャノン情報量の定義式と同じであるが，シャノ

ン情報量の確率分布が情報源の確率分布から定義されているのに対し，経験エントロピーは実際に観測された文字列中の出現確率（経験確率）で与えられている点が異なる．この 0 次経験エントロピーは，各文字を出現確率の情報のみを用いて符号化した場合の，符号の平均長の下限と考えることもできる．

すべての文字が同じ割合で出現する場合，経験エントロピーは最大である $H_0(T)=\lg\sigma$ をとり，逆に，出現割合の偏りが大きいほど小さい値をとる．例えば，文字列 $T=$“*abccbaabca*” の場合，$p(a)=4/10=0.4$，$p(b)=3/10=0.3$，$p(c)=3/10=0.3$，$H_0(T)$ は約 1.57 であり，すべて同じ文字 “*a*” からなる文字列 $T=$“$aaa\cdots aaa$” の場合，$H_0(T)=0$ になる．

次に，k 次経験エントロピーを次のように定義する．

定義 2.3（k 次経験エントロピー）

$$H_k(T) = -\sum_{W\in\Sigma^k}\frac{n_W}{n}H_0(T_W) \tag{2.3}$$

□

ただし，T_W は文字列 T 中に出現する部分文字列 W の直前に出現する文字をつなげて構成された文字列であり，n_W は，文字列 T 中で部分文字列 W が出現した回数である．ただし，T の先頭に対応する T_W については考えない．

例えば，文字列 $T=$“*abccbaabca*” のとき，$T_a=$“*bac*”，$T_b=$“*aca*”，$T_c=$“*bcb*” であり，$H_0(T_a)=1.58$，$H_0(T_b)=0.92$，$H_0(T_c)=0.92$ であり，$H_1(T)=1.14$ である．k 次経験エントロピーは文字列 T の各文字を符号化する際，その直後に後続する長さ k の部分文字列を利用して符号化した場合の，符号の平均長の下限である．

この経験エントロピーについては次のような階層構造が成り立つ．

$$H_k(T) \le H_{k-1}(T) \le \cdots \le H_0(T) \le \lg\sigma \tag{2.4}$$

つまり，より多くの文脈情報を利用するほど，より正確に次の文字を予測できる．例えば，一般的な英語のテキストの場合 $\lg\sigma=8$, $H_0(T)=4.5$, $H_5(T)=1.8$ 程度であることが報告されている．

これから紹介する多くのデータ構造やデータ圧縮は，この経験エントロピーに近いサイズでデータを格納することが 1 つのゴールとなる．しかし，経験

エントロピーを達成できれば達成可能な最小サイズになっているかといえば，必ずしもそうではない．k 次経験エントロピーでは，k 文字以内の文字間の関係をとらえることができたが，それ以外の特徴をとらえることができない．例えば，長い文字列の繰り返しがある場合でも，経験エントロピーはその冗長性をとらえることができない．この他にも，文字列中で一度出現した文字列は再度出現しやすいといった情報も，とらえることはできない．

これを見るために，$|S|>k$ であるような文字列 S が u 回繰り返される文字列 $T=SSS\cdots SSS$ を考えてみる．この場合，T の k 次経験エントロピーは S の k 次経験エントロピーとほとんど変わらない．なぜなら，S でも T でも，$k-1$ 文字の直前の文字の出現分布は変わらないからである．しかし，T は S の繰り返しとして表すことができれば，1 文字あたり，$H_k(S)/u$ に近いビット数で表現でき，k 次経験エントロピーによるビット数と大きな差がみられる．

こうした制限があるものの，経験エントロピーは文字列の持つ冗長性や各表現の効率性を示すよい指標として多く使われる．

2.4　計算モデル

本書ではさまざまなアルゴリズムを紹介するが，その性能は抽象化された計算機上での計算回数を測って表現したい．この章では抽象化された計算モデルを単純なものから順に紹介する．

最も単純な Cell Probe モデルでは，メモリへの参照(読み込み，書き込み)に対してのみコストがかかり，それ以外の計算コストは無視する．メモリはセルの集合からなり，各セルのサイズはモデルのパラメータである w で決められる．このモデルは現実世界のモデルとは離れているが，計算量の下限が計算しやすい．

Cell Probe モデルの中でも，対象データのサイズが n のときモデルのパラメータ w が $w=\lg n$ であることを仮定したのが Transdichotomous RAM モデルである．つまり，問題のサイズに応じて計算モデルが変化する．このモデルではデータが与えられたとき，データ中のポインタや添字番号が定数個のセル

で格納できる．

Word RAM モデルは Transdichotomous RAM モデルに加え，サイズが $O(w)$ であるデータに対する任意の操作(四則演算など)に対する操作回数もコストとしてカウントしたモデルである．Word RAM モデルは現在最も使用されているモデルであり，実際の計算機のモデルに最も近いモデルである．

本書では Word RAM モデルを扱い，この計算機上での計算コストで各手法の計算量を評価する．

2.5 符　号

次に，データを計算機上でどのように表現するかについて考えてみよう．与えられた文字列をビット列に変換することを符号化と呼び，逆にビット列からもとのデータに変換することを復号化と呼ぶ．そして，符号化，復号化のルールをひとまとめにして符号法と呼ぶ．

こうした符号法の中で，すべての文字列に対して常に符号化した文字列を復号化して，誤りなくもとの文字列が得られる場合，この符号法は可逆であると呼ぶ．可逆な符号法では，一部のデータを小さく表現することはできるが，すべての文字列を同時に小さく圧縮はできない．つまり，どんなデータでも必ず圧縮できる符号法は存在しない．これを示すために，長さ k のあらゆるビット列を符号化した結果，長さ $t<k$ のビット列が得られたと仮定する．すると，もともと 2^k 通り存在したビット列が 2^t 通りのビット列に変換されたことになるので，一対一の対応関係をとることはできず，この符号化は可逆ではない．

よって，符号を選んでデータを小さくするデータ圧縮は“賭け”だと考えてよい．よく出現しそうな文字列に対してより短い符号を割り当て，その代わりに他の出現しなさそうな文字列に対しては，より長い符号を割り当てる．これにより全体の符号長を短くできる．世の中のデータには偏りが多くみられるので，こうした偏りを見つけ出し，適切な表現方法をとることで，もとの文字列の情報を失うことがないまま全体長を短くできる．

次に，文字列の符号化を各文字の符号をつなげて構成した場合を考えてみる．各文字の符号長がすべて同じ場合は，各文字の符号がどこから始まってい

るかがわかる．しかし，各文字の符号長が異なる場合は，一般には各文字の符号がどこから開始しているかわからず，もとの文字列を一意に復元できない．

前から順に読み込んで復号を決定的に行うことができるような符号法を瞬時復元可能な符号法と呼び，瞬時復元可能な符号として，接頭辞符号が知られている．

定義 2.4(接頭辞符号)　もとの入力文字列のアルファベット $\{x_1, x_2, \cdots, x_k\}$ それぞれに対し，符号を割り当てたとき，ある文字の符号が，他の文字の符号の接頭辞になっていないような符号を接頭辞符号とよぶ．　□

例えば，符号集合 $\{0, 01, 11\}$ は接頭辞符号ではない．なぜなら，0 は 01 の接頭辞だからである．それに対し，$\{0, 10, 11\}$ は接頭辞符号である．接頭辞符号は葉に文字が割り当てられた二分木で表すことができる．

2.5.1　単進符号，二進符号

以降では，可算無限個の正整数 $\{1, 2, \cdots\}$ に対し符号を割り当てることを考える．

正整数 x の単進表現(Unary 表現)を $(x)_1$ とする．これは $x-1$ 回，0 を繰り返した後に 1 をつなげたビット列からなる．例えば $(1)_1=1, (5)_1=00001, (10)_1=0000000001$ である．正整数 x の二進表現(Binary 表現)を $(x)_2$ とする．例えば $(1)_2=1$，$(5)_2=101$，$(777)_2=1100001001$ である．

単進符号はどの符号も他の符号の接頭辞になっていないため，接頭辞符号である．これに対し，二進符号は接頭辞符号ではない．例えば 11111 という符号があったときに，この符号はもともと 1,1,1,1,1 なのか 11,11,1 なのかわからない．また，単進符号は接頭辞符号であるが，値が大きい場合に符号長が長くなってしまう問題がある．これに対し二進符号は，接頭辞符号ではないが，値が大きくても符号長は値の対数に比例する程度しか増加しないという特徴がある．

単進符号と二進符号を組み合わせることで，さまざまな特徴を備えた符号が構成できる．これらを以下に見ていく．

2.5.2 γ 符 号

二進表現の問題は接頭辞符号ではなく，各文字の符号の区切り目がわからないことであった．そこで二進表現の符号長を単進符号で表し，それに続いて二進表現を記録する．

この符号化の際に，二進表現では最上位ビットが常に 1 であることから，それを省略する．符号 x の最上位ビットを除いた符号を $P(x)$ と書くことにする．例えば，$P(1100_2){=}100_2$ である．文字 x に対する符号 $f(x)$ に対し，その符号長を $|f(x)|$ で表す．このとき，正整数 x を $\gamma(x){=}(|(x)_2|)_1 P((x)_2)$ として表現する符号を γ 符号と呼ぶ．

この γ 符号は接頭辞符号である．これを示すために，異なる正整数 $x{<}y$ それぞれに対する γ 符号を考えたとき，$|\gamma(x)|{\leq}|\gamma(y)|$ なので，x が y の接頭辞と一致しないことを示せばよい．$|(x)_2|{<}|(y)_2|$ ならば，単進符号が接頭辞符号であることから，x は y の接頭辞ではない．$|(x)_2|{=}|(y)_2|$ のとき，$(x)_2{\neq}(y)_2$ であるので接頭辞は一致しない．よって示された．

γ 符号の符号長は $2\lfloor\lg x\rfloor{+}1$ である．

2.5.3 δ 符 号

γ 符号の符号長は，二進表現の約 2 倍であった．そのため，値が大きい場合に二進表現に比べ，符号長がかなり長くなってしまう．そこで γ 符号の中で単進符号で長さを表している部分をもう一度 γ 符号で表した表現を考える．つまり，正整数 x を $\gamma(|(x)_2|)P(x)_2$ として表現する符号を δ 符号と呼ぶ．δ 符号の符号長は $\lfloor\lg x\rfloor{+}\mathrm{o}(\lfloor\lg\rfloor)$ ビットである．

δ 符号は γ 符号に対し，小さい値に対しては符号が長く，大きい値に対しては符号は短くなる．

2.5.4 ゴロム符号・ライス符号

正整数 x に対し，$x'{=}x{-}1$ とする．また，正整数パラメータ $k{>}0$ を用意し，$r{=}(x'/k){+}1$，$m{=}x' \bmod k$ とする．そして，r を単進符号，m を長さ $|(k)_2|$ ビットの二進符号で記録し，これらをつなげた符号をゴロム符号と呼ぶ．m

は $0 \le m < k$ なので，常に $|(k)_2|$ ビットで記録できることに注意する．

さらに，k が 2 のベキ乗の場合のゴロム符号をライス符号と呼ぶ．ライス符号の場合は r, m をそれぞれビットシフト操作，ビットマスク操作で得られるため高速に符号化できる．ゴロム符号，ライス符号は接頭辞符号である．

2.5.5 ハフマン符号

ハフマン符号は，それぞれの文字に対して符号を割り当てたときに全体長が最小となるような符号の割り当て方である．これまでの符号とは違い，ハフマン符号では対象文字列によって符号が変わる．よって，符号側，復号側が出現確率表もしくは符号を共有する必要がある．

ハフマン符号では，まずハフマン木を次のように構築する．各文字ごとにその出現回数を重みとした葉を考える．次に，重みが最小である節点(葉を含む) a, b を 2 つ選び，それらを併合して新しい節点 c を作り，a, b の重みの和を c の重みとする．このとき，a, b はそれぞれ c の左の子，右の子となる．この操作を節点の数が 1 個になるまで繰り返し，最終的には各文字が葉であるような二分木ができる．これがハフマン木である．このとき，文字 x の符号は根から x に対応する葉に向かって木のパスをたどり，左の子に進んだときは 0, 右の子に進んだときは 1 をつなげたビット列で与えられる．

この他にもハフマン符号の符号化，復号化の高速化するために木を使わずに表引きのみで符号化，復号化が行える標準化ハフマン符号も知られている [20].

2.5.6 符号間の比較

表 2.2 に，各符号による正整数の符号例を示す(出現確率に依存するハフマン符号は省略する)．1 つの符号が常に優れているわけではなく，整数の出現確率の分布によって最適な符号が変わってくる．どのような場合に最適になるのかを以下に見てみよう．ある文字 x の出現確率が p の場合，その文字の符号長が $\lg p$ ビットであれば最適である．逆に，文字 x の符号長が l であればその出現確率は 2^{-l} のときに最適である．

単進符号の符号長は正整数 x に対し，x ビットであった．よって各整数 x

表 2.2　正整数に対する各種符号による比較

	Unary	Binary	γ	δ	ゴロム(r=8)
1	1	1	1	1	1000
2	01	10	010	0100	1001
3	001	11	011	0101	1010
4	0001	100	00100	01100	1011
5	00001	101	00101	01101	1100
10	0000000001	1010	0001010	00100010	01001
100	000...0001	1100100	0000001100100	00111100100	0000000000001011

の出現確率 $p(x)$ が $x=-\lg p(x)$ を満たすとき，つまり，$p(x)=\dfrac{1}{2^x}$ であるとき，単進符号は最適となる．具体的には $p(x=1)=1/2$，$p(x=2)=1/4$，$p(x=3)=1/8$ と，大きい値の出現確率が急速に小さくなる場合に単進符号は平均符号長が最小となる．

同様に γ 符号は出現確率 $p(x)$ が $2\lfloor\lg x\rfloor+1=-\lg p(x)$ を満たすときに最適となる．単純化のために，先ほどの式から $\lfloor\ \rfloor$ を除いた場合を考えると，$p(x)=\dfrac{1}{2x^2}$ のときとなる．単進符号よりはずいぶんゆっくりと確率が減少する場合に γ 符号は平均符号長が最小となる．ゴロム符号の場合はパラメータが k であれば，$x/k+k=-\lg p(x)$ の場合，$p(x)=\dfrac{1}{2^{x/k+k}}$ のときに最小を達成する．ハフマン符号は，すべての文字の出現確率がわかっている場合には，全体長が最小になる最適な符号である．復元側は，圧縮時に利用したハフマン木を知っている必要があり，圧縮側と復元側がハフマン木を共有できる場合，または符号化対象のデータがハフマン木のサイズより十分に大きい場合に有効である．

3

Burrows Wheeler 変換

Burrows Wheeler 変換(BWT)は文字列解析をするうえで重要なツールである．BWT を用いることで，文字列中に含まれる部分文字列の情報を効率的に表現でき，部分文字列の頻度などさまざまな統計量を高速で計算できるほか，文字列自体を効率的に格納できる．こうした特徴から BWT はデータ圧縮，全文索引，データマイニングなど多くの用途で利用される．本章では，BWT とは何か，そして BWT をどのように実現するかを紹介する．

3.1 接尾辞配列

はじめに，文字列を扱ううえで重要なデータ構造である，接尾辞配列を紹介する．

定義 3.1（接尾辞配列） 文字列 $T[0,n)$ が与えられたとする，ただしその末尾 $T[n-1]=\$$ は他には出現せず，どの文字よりも小さな文字とする．この T のすべての接尾辞 $S_i=T[i,n)$, $0 \leq i<n$ を辞書式順序でソートし，得られた順序が $S_{i_1}<_{lex}S_{i_2}<_{lex}\cdots<_{lex}S_{i_n}$ であったとする．このときの添字番号 $i_1, i_2, \cdots, i_n$ を格納した配列を接尾辞配列(Suffix Array)と呼び，$SA_T[0,n)$ で表す． □

このとき，T の末尾に特殊文字($T[n-1]=\$$)があるおかげで，すべての接尾辞間で辞書式順序が定義できることに注意する．以降，曖昧性がない場合，SA_T の添字を省略し SA と表す．

先ほどの定義を別の言い方をすると，文書のすべての位置から始まる部分文字列を辞書式順序で小さい順に並べ，その位置を格納したものが接尾辞配列である．すべての位置の接尾辞について並び替えているので，接尾辞配列は $\{0,1,2,\cdots,n-1\}$ を並び替えた配列である．図 3.1 に $T=$“$abracadabra\$$” に対する接尾辞配列の例を示す．接尾辞配列のサイズは，各要素が $\lg n$ ビット必要なので，全体で $n \lg n$ ビットである．

接尾辞配列を利用して T 中の任意の文字列を効率よく検索するための重要な性質を述べる．

補題 3.2 T に出現するすべての部分文字列は，接尾辞配列上では連続する接尾辞中の接頭辞として出現し，それらの出現位置は SA 中の範囲 $SA[sp,ep)$ として表現できる． □

例えば，図 3.1 の中で，部分文字列 “$abra$” は，$SA[2,4)=[7,0]$ の接尾辞(S_7, S_0)の接頭辞として出現し，“a” は $SA[1,6)=[10,7,0,3,5]$ の接尾辞(S_{10}, S_7, S_0, S_3, S_5)の接頭辞として出現している．これを利用し，部分文字列が T 中のどこに出現していたかは，接尾辞配列上のどの範囲 $SA[sp,ep)$ で連続して出現していたかを調べればよい．

接尾辞配列上では，接尾辞は辞書式順序でソートされているので，長さ m

$T = abracadabra\$$

0	$abracadabra\$$
1	$bracadabra\$$
2	$racadabra\$$
3	$acadabra\$$
4	$cadabra\$$
5	$adabra\$$
6	$dabra\$$
7	$abra\$$
8	$bra\$$
9	$ra\$$
10	$a\$$
11	$\$$

⇨

i	SA	T_B	接尾辞
0	11	a	$\$$
1	10	r	$a\$$
2	7	d	$abra\$$
3	0	$\$$	$abracadabra\$$
4	3	r	$acadabra\$$
5	5	c	$adabra\$$
6	8	a	$bra\$$
7	1	a	$bracadabra\$$
8	4	a	$cadabra\$$
9	6	a	$dabra\$$
10	9	b	$ra\$$
11	2	b	$racadabra\$$

図 3.1 $T=$“$abracadabra\$$” に対する接尾辞配列と BWT の例．列 SA に接尾辞配列，列 T_B に BWT 配列を示す．

のクエリ文字列 $Q[0,m)$ が出現している接尾辞配列の範囲 $SA[sp,ep)$ は接尾辞配列上の二分探索により $O(m\log n)$ 時間で求めることができる．詳しくは第 7 章で取り扱う．

この計算時間は部分文字列の出現回数によらないことに注意する．またこの範囲がわかれば部分文字列の出現回数は $ep-sp$ で求められ，出現位置は $SA[sp,ep)$ によって与えられる．

3.2 Burrows Wheeler 変換

Burrows Wheeler 変換（BWT, Burrows Wheeler Transform）は Burrows, Wheeler らによって提案された文字列の可逆変換である．BWT は 1980 年頃から Wheeler によって考案されていたが，彼自身が「自明であり，発表するほどでもない」とし外部発表されなかった．しかし，その重要さに気づいた

Burrows によって後ほど Block Sorting という名前でテクニカルレポートにまとめられ，世の中に発表された．BWT はもともと暗号化やデータ圧縮のために考案されたが，その後，全文検索やデータマイニングでも非常に有用なツールであることがわかり，関連する研究分野は非常に広がっている．

BWT の定義を示す．

定義 3.3（Burrows Wheeler 変換（BWT）） 文字列 $T[0,n)$ とその接尾辞配列 $SA[0,n)$ が与えられたとき，BWT は次のように定義される．

$$T_B[i] = \begin{cases} T[SA[i]-1] & SA[i] > 0 \\ T[n-1] & SA[i] = 0 \end{cases} \tag{3.1}$$

□

文字列 T の BWT の結果の文字列 T_B を T の BWT と呼ぶことにする．

この定義は次のように言い換えることができる．入力文字列 $T[0,n)$ に対し，各文字 $T[i]$ を，それに後続する接尾辞 $T[i+1,n)$ をキーとした辞書式順序で並び替える操作が BWT である，ただし，末尾の文字 $T[n-1]$ のキーは $T[0,n)$ とする．また T_B は T を並び替えた文字列である．このことより，例えば，T 中の各文字の頻度と T_B 中の各文字の頻度は 1 対 1 の並び替えになっていることより一致するといった簡単な事実は導きだされる．

図 3.1 の列 T_B に T=“*abracadabra*\$” の場合の BWT の例を示した．この T に対する BWT は T_B=“*ard*\$*rcaaaabb*” となる．例えば，辞書式順序で最初の接尾辞は S_n=“\$” であるので，$T[n-1]$=“*a*” が T_B の最初の文字となり，S_{n-1}=“*a*\$” が 2 番目に小さいので $T_B[1]$=$T[n-2]$=“*r*” となる．

BWT は文字列をより圧縮しやすい文字列に変換する．図 3.2 に一般的な英文に BWT を適用したときの結果を示す．このように BWT を適用した後の文字列は同じ文字が連続して出現しやすくなる．なぜこのような現象が起きるかというと，まず BWT は各文字を後続する接尾辞の辞書式順序をキーとしてソートする．このとき，後続している文字列が似ている場合，同じ文字が直前に出現しやすい．例えば，英語では “The”という文字が頻出するが，この場合 “he …”という接尾辞の直前には “T”が出現しやすいので，BWT の中で接尾辞が “he …”に対応する領域では T が出現しやすくなる．別の言い方をす

London Bridge is broken down, Broken down,
broken down. London Bridge is broken down,
My fair lady. Build it up with wood and clay,
Wood and clay, wood and clay, Build it up with
wood and clay, My fair lady. Wood and clay
will wash away, Wash away, wash away, Wood
and clay will wash away, My fair lady



.nn,,..,,,,,., ddddddhhhhss, ddddddnnnnyyyeedd
rrrttll, yypp, hhnyynyyyynyyyyny   lllfff
wwwW   lllwlwwlwl   oooooonnnnnnlliinn
aaaggkkkk   ddsssstttrruuwwaaa   wwooooll
ccccccciiiiooeeeewwwwaaaaaaoooooooorrrrddL
LWWwwwWdddduuiiiBBbbBbiiaaaaiiiiBB
aaaa   oooo   MMMaaaaaaaaaaddd

図 3.2 BWT による変換例．BWT は元の文字列を並び替えるだけだが，変換後は同じ文字が連続して出現しやすい文字列に変換される．

れば文字列の k 次経験エントロピーが低い場合，その BWT では同じような文字が連続しやすいといえる．この文字列をどのように圧縮するか，またその性能はどのようなものかについては 6.4 節で説明する．

同じ情報を違う形で表現することで，効率的に表現できたり操作を高速に実現できるという点で，BWT はフーリエ変換と似ている．フーリエ変換では，時間情報と波の強さという軸で表される波形情報を各周波数ごとの正弦波，余弦波の足しあわせという形に変換し表現する．この変換により，もとの表現では困難であった周波数ごとの演算や畳込み演算などを効率的に行うことができる．同様に BWT では，各文字が文字列中の位置順に記録されていたのを，各文字に後続する辞書式順序で並び替えて変換し表現する．この変換により BWT では，部分文字列情報を効率的に扱うことができる．

非常に重要な点は BWT は変換後の文字列の情報のみからもとの文字列を復元できることである．これ以降では，文字列 T の BWT である T_B のみが与えられた状況で，T を復元する方法を説明する．また，このときに利用す

る BWT の性質は文字列解析でも利用する．

接尾辞配列順に並んだ各接尾辞の先頭文字を $T_F[0,n)$, $T_F[i]{=}T[SA[i]]$ と定義する．T_F は T_B のみから，次のように得ることができる．まず，$C[c]$ が $c{\in}\Sigma$ より小さい文字の T 中の出現回数であるような配列 $C[0,\sigma)$ を考える．BWT 後の文字列 T_B は T を並び替えて構成された文字列であることを考慮すると，T_B 中の各文字の出現回数と，T の出現回数は一致するので，C は T_B 中の各文字の出現回数を求めることにより得られる．また辞書式順序の定義より，$T_F[0,n)$ は T 中の文字が小さい順に並んでいることがわかる．よって，T_F は，$T_F[C[c]]$ と $T_F[C[c{+}1]{-}1]$ の間はすべて c ということになる．式で書けば $T_F[C[c],C[c{+}1]){=}$“$cc{\cdots}c$” である．例えば，先ほどの例では $T_F{=}$“\$$aaaaabcdrr$” となり，$C$[“$a$”]=1, C[“b”]=6, C[“c”]=7 である．

次に，$T_B[i]$ と $T_F[i]$ はそれぞれ，$T[SA[i]{-}1]$，$T[SA[i]]$ に対応するので，$T_B[i]$ と $T_F[i]$ はもとの文字列 T 中では隣り合って出現していたことがわかる．しかし，これらの文字がもとの文字列中のどの位置で出現していたかはまだわからない．

ここで各文字がどのように対応しているのかについて見てみる．文字列 $T{=}$“$abracadabra\$$” に対し，その BWT は $T_B{=}$“$ard\$caaaabb$” であるが，各文字を区別するために，$T_B{=}$“$a_1r_1d_1\$c_1a_2a_3a_4a_5b_1b_2$” のように番号をつけたとする．例えば，“$a_2$” は T_B 中で前から 2 番目の “a” ということを意味する．このとき，T_F ではこれらの文字は $T_F{=}$“$\$a_1a_2a_3a_4a_5b_1b_2c_1d_1r_1r_2$” のように出現しており，各文字ごとの出現順序は変わらない．これは偶然ではなく常に成り立つ．

補題 3.4　各文字 c に着目した場合 T_B 中の出現順序と T_F 中の出現順序は一致する．□

［証明］ $T_F[i]{=}T_F[j]{=}c,\ i{<}j$ である文字について注目してみる．辞書式順序の定義より $S_{SA[i]}{<}S_{SA[j]}$ が成り立つ．さらに，$S_{SA[i]}, S_{SA[j]}$ は先頭文字が c で同じであり，辞書式順序の定義より 1 文字めが同じ場合は 2 文字め以降の比較結果で順序が決まっているはずなので，$S_{SA[i]+1}{<}S_{SA[j]+1}$ が成り立つ．さて，$T_F[i], T_F[j]$ に対応する文字が，T_B で出現している位置を考えると，各文字はそれらに後続する接尾辞，つまり $S_{SA[i]+1}, S_{SA[j]+1}$ の間で比較がされ

i	T_B	接尾辞(先頭文字が F)
0	a_1	\$
1	r	a_1\$
2	d	a_2bra\$
3	\$	$a_3bracadabra$\$
4	r	$a_4cadabra$\$
5	c	a_5dabra\$
6	a_2	bra\$
7	a_3	$bracadabra$\$
8	a_4	$cadabra$\$
9	a_5	$dabra$\$
10	b	ra\$
11	b	$racadabra$\$

i	T_B	接尾辞(先頭文字が F)
0	a_1	
1		a_1
2		a_2bra\$
3		$a_3bracadabra$\$
4		a_4
5		a_5
6	a_2	bra\$
7	a_3	$bracadabra$\$
8	a_4	
9	a_5	
10		
11		

比較文字は同じ

図 3.3　同じ文字において，BWT 中の文字の出現順序と，T_F 中の文字の出現順序は比較に利用する文字列が同じため，一致する．

て辞書式順序に並べられているので，$T_F[i], T_F[j]$ の出現順序は T_B 中でも変わらない．(図 3.3 の例)． ■

こうして T_B 中の文字と T_F 中の文字がすべて対応付けられることがわかった．あとは，先頭の文字から順に対応付けを参考に復元を行えばよい．先頭の位置は $T_B[i]=\$$ である次の場所として見つけることができる．例えば先ほどの例では，“a_3” が次の文字であった．“a_3” が T_B で出現しているのは i=7 であり，次の文字は “b_2” である．これを繰り返していけば，もとの文字列 “$a_3b_2r_2a_4c_1a_5d_1a_2b_1r_1a_1\$$” が復元できる．図 3.3 の例で実際に試してほしい．この復元を効率的に実現する手順については 3.4 節で述べる．

3.3　接尾辞配列，BWT の構築

接尾辞配列や BWT は定義の単純さとは違って，単純な構築方法では計算量が大きいという問題がある．接尾辞配列や BWT を効率よく構築する方法について紹介する．ただし，この節は高度な内容を含むので，読み飛ばしてもらっても構わない．

3.3.1 接尾辞配列の構築

文字列 $T[0,n)$ に対する接尾辞配列を構築するアルゴリズムを考えてみる．

はじめに，定義通りの方法，つまり文字列 $T[0,n)$ 中のすべての接尾辞を辞書式順序にしたがってソートし接尾辞配列を構築する方法について考えてみよう．接尾辞は n 個あるのでソート時の接尾辞間の比較回数は $O(n \log n)$ 回である．しかし，例えば，入力文字列が同じ文字だけからなる場合($T=$ "$aaaa \cdots aaa$")，接尾辞間の比較が毎回 $O(n)$ 回必要となり全体の計算量は $O(n^2 \log n)$ となる．n が大きい場合に，$O(n^2)$ という項は非常に大きくなるため，現実的には処理は不可能であることを意味する．非常に大きな文字列に対して接尾辞配列を構築するには $O(n \log n)$ 時間や $O(n)$ 時間といった計算量で処理できるアルゴリズムが求められる．

3.3.2 Induced Sorting

接尾辞配列を効率的に構築するさまざまなアルゴリズムが提案されてきた．ここではその中で理論的にも実用的にも優れている Induced Sorting を紹介する[25]．

Induced Sorting は，アルファベットのサイズによらず，長さ n の文字列に対し常に $O(n)$ 時間で接尾辞配列を構築できる．さらに作業領域量は構築対象のテキストのサイズ $n \lg \sigma$ ビットと接尾辞配列のサイズ $n \lg n$ ビットの合計と一致する．

以降では接尾辞 $S_0 S_1 \cdots S_{n-1}$ を扱う．このときの接尾辞の順序は辞書式順序ではなく，もとのテキストの順序であり，$S_i = T[i,n)$ である．

定義 3.5(接尾辞の Type)　すべての接尾辞 $\{S_i\}_{i=0,\cdots,n-1}$ を次の 2 つの Type に分類する．

- Type-S: S_{n-1} または $S_i < S_{i+1}$ を満たす接尾辞．
- Type-L: $S_i > S_{i+1}$ を満たす接尾辞．

□

例えば，$T=$ "*abracadabra\$*" に対し S_0 は，$S_0=$ "*abracadabara*" $< S_1=$ "*bracadabara*" であるので S_0 は Type-S である．

これらのタイプは次のように後ろから順に調べて O(n) 時間で決定できる．

- S_{n-1} は Type-S である．
- S_i のタイプは $T[i]<T[i+1]$ ならば Type-S，$T[i]>T[i+1]$ ならば Type-L，$T[i]=T[i+1]$ ならば，S_{i+1} のタイプと同じである．

また，Type-S であるような接尾辞 S_i のうち，S_{i-1} が Type-L であるような接尾辞を Type-S^* と呼ぶことにする．なお，S_{n-1} は Type-S^* であるとする．

接尾辞のタイプを順に並べてみると，$LLLS^*SSSLLLS^*SSSLLLS^*$ というように Type-S と Type-L が交互に並び，S^* は連続する S の先頭に位置する．図 3.4 の上に，$T=$"$abracadabra\$$" に対する Type を示す．

タイプについて次が成り立つ．

補題 3.6 Type-S^* である接尾辞は高々 $\lceil n/2 \rceil$ 個である． □

また，タイプ間の辞書式順序については次が成り立つ．

補題 3.7 Type-S である接尾辞 S_i と Type-L である接尾辞 S_j について，これらの先頭文字が同じ場合，つまり $T[i]=T[j]$ のとき，$S_i>S_j$ が成り立つ． □

［証明］ 辞書式順序の定義より，S_i と S_j の辞書式順序は先頭文字が同じことから 2 文字め以降である S_{i+1} と S_{j+1} との間で比較が行われる．S_{i+1} は定義より S_i より大きいので $y=T[i]$ としたとき，S_{i+1} は y が 0 回以上繰り返された後に y より大きい文字 z が後続する形をとるはずである．

それに対し，S_{j+1} は y が 0 回以上繰り返された後に y より小さい文字 x が後続する形をとる．よって，S_{i+1} と S_{j+1} を比較したとき，常に，S_{i+1} の方が大きく，$S_i>S_j$ が成り立つ． ■

この事実により，接尾辞配列において，先頭文字が同じである接尾辞の中では最初に Type-L の接尾辞が並び，その後に Type-S の接尾辞が並ぶことになる．

Induced Sorting は次のようにして接尾辞配列を構築する．

- Step1: Type-S^* の接尾辞をソートする．
- Step2: Type-S^* の順序から Type-L の接尾辞の順序を決定する．
- Step3: Type-L の順序から Type-S の接尾辞の順序を決定する．

各ステップはいずれも O(n) 時間で実行可能である．最初にソート済みである Type-S^* が得られたと仮定し，ステップ 2 とステップ 3 がどのように求め

	0	1	2	3	4	5	6	7	8	9	10	11
T	a	b	r	a	c	a	d	a	b	r	a	\$
Type	S	S	L	S*	L	S*	L	S*	S	L	L	S*

i	T_B	接尾辞	
0	a	\$	\$
1	**r**	**a\$**	a
2			a
3	d	$abra$\$	a
4	r	$acadabra$\$	a
5	c	$adabra$\$	a
6			b
7			b
8			c
9			d
10			r
11			r

i=0, … と順に調べ S_{i-1} が Type-L である場合，c=$T[i-1]$ のバケットの先頭に S_{i-1} を登録する

i	T_B	接尾辞	
0			\$
1	r	a\$	a
2			a
3			a
4			a
5			a
6			b
7	**a**	**$bracadabra$\$**	b
8	a	$cadabra$	c
9	a	$dabra$\$	d
10	b	ra	r
11	b	$racadabra$\$	r

i=$n-1$, …, 0 と調べ S_{i-1} が Type-S である場合，c=$T[i-1]$ のバケットの先頭に S_{i-1} を登録する

図 3.4 線形時間での接尾辞配列構築

られるかについて見ていく．図 3.4 に，各ステップの例を示す．

Type-S^* から Type-L の接尾辞の順序を決定する

ソート済みの Type-S^* の接尾辞が与えられたとき，Type-L の辞書式順序を決定する．はじめに，$C[0,\sigma]$ を $C[i]$ が T 中の文字 i よりも小さい文字の出現回数であるような配列とする．文字 $T_F[i]$=c である接尾辞の範囲を c のバケットと定義する．この範囲は $[C[c], C[c+1])$ として求められる．ソート済みの Type-S^* の配列を前から順に調べ，各 S_i について，もし S_{i-1} が Type-L であれば，c=$T[i-1]$ から始まる接尾辞の中で S_{i-1} は現在決定していない接尾辞の中で最も小さいので，S_{i-1} を c のバケットの最後の位置である $C[c]$ に $SA[C[c]]=i-1$ と格納し，$C[c]:=C[c]+1$ と更新する．また，すでに決まった各 Type-L の接尾辞についても，その直前の接尾辞 S_{i-1} が Type-L であれば，それは現在決定していない接尾辞の中では最も小さいはずであるので，同様に接尾辞の順位を決定する．

Type-L から Type-S の接尾辞の順序を決定する

次にソート済みの Type-L の接尾辞が与えられたとき，Type-S の辞書式順

序を決定する．$C[0,\sigma)$ を $C[i]$ が T 中の文字 i よりも小さい文字，もしくは同じ文字の出現回数を保持したような配列とする．Type-L である接尾辞を後ろから順に調べていき，もし S_i について S_{i-1} が Type-S であれば，S_{i-1} は現在決定していない接尾辞の中で最も大きいので，$c{=}T[i{-}1]$ のとき，$C[c]$ の位置に S_{i-1} を登録し，$C[c]{:=}C[c]{-}1$ と更新する．また，すでに決まった各 Type-S の接尾辞についても，その直前の接尾辞 S_{i-1} が Type-S であれば，同様に S_{i-1} の接尾辞の順位は決定される．

上記 2 つのステップにより，すべての接尾辞の順序が決定される．では最初の Type-S^* の接尾辞集合間の順序決定をどのように行うかについて，以下に述べる．

Type-S^* の順序を決定する

接尾辞 S_i が Type-$S/L/S^*$ である場合，文字 $T[i]$ も同様に Type-$S/L/S^*$ であると呼ぶことにする．

先頭と最後の文字が Type-S^* であり，それ以外は Type-S^* の文字を含まないような部分文字列を S^* 部分文字列と呼ぶ．ただし，最後の文字はそれのみからなる S^* 部分文字列とする．S^* 部分文字列は，Type の並びで見ると Type-S^* の文字で開始し，0 回以上の Type-S，1 回以上の Type-L，そして最後が Type-S^* で終わるような文字列である．例えば "*abracadabra*\$" の場合の S^* 部分文字列は，"*aca*", "*ada*", "*abra*\$", "\$" の 4 つからなる．各 S^* 部分文字列においては最初と最後の Type-S^* 文字が重複する．

Type-S^* の接尾辞を S^* 部分文字列をつなげた配列として考えてみる．このときもとの接尾辞間の順序と S^* 部分文字列をつなげた文字列の順序は同じである．各 S^* 部分文字列の順序を決定し，その順序番号を S^* 部分文字列に置き換える．T="*abracadabra*\$" の Type-S^* である接尾辞は S_3="*acadabra*\$", S_5="*adabra*\$", S_7="*abra*\$", S_{11}="\$" の 4 つである．T 中の S^* 部分文字列の順序を決定すると "\$"=0, "*abra*\$"=1, "*aca*"=2, "*ada*"=3 となり，Type-S^* である接尾辞は S_3=2310, S_5=310, S_7=10, S_{11}=0 に変換される．

各 S^* 部分文字列の辞書式順序は Radix ソートと呼ばれるものを利用し，合計 $O(n)$ 時間で求めることができる．この処理は実は先ほどの Step-2，Step-3 の方法とまったく同じ操作で実現できる．

さて，このような変換でもとの文字列 T は S^* 部分文字列の順位がつながった新しい文字列 T_2 に変換され，新しいアルファベット集合が得られた．S^* 部分文字列の数が $n/2$ 以下であることから T_2 の長さは $n/2$ 以下である．新しくできた接尾辞の順位は T_2 に対して再帰的に同じアルゴリズムを利用して求めればよい．

この再帰はすべての文字が異なるアルファベットになったときに終了する．このアルゴリズムの全体の計算量は文字列の長さに対し線形である．また，必要な作業領域量はもとの文字列に加えて，接尾辞配列を格納する $n \lg n$ ビットである．

3.3.3 BWT の構築

BWT の構築については，接尾辞配列 SA を構築した後に，$T_B[i]=T[SA[i]-1]$ の定義にしたがって BWT を構築する方法，および接尾辞配列を構築せずに直接 BWT を求める方法がある．

接尾辞配列を介して構築する方法は直前の章で扱った方法を利用すれば，O(n) 時間で構築でき，その際の作業領域量は接尾辞配列の $n \lg n$ ビットと元テキストと BWT の $2n \lg \sigma$ ビットの合計である．

しかし，BWT のサイズは $n \lg \sigma$ ビットであり，接尾辞配列のサイズの $n \lg n$ ビットと比べると，アルファベットサイズが小さい場合は特に小さい．例えば，DNA 配列の場合は $\sigma=4$, $\lg \sigma=2$ であり，$\lg n=30$ といった場合と比べて 1/10 以下である．

本章では接尾辞配列を構築せずに，直接 BWT を構築する方法について紹介する．これにより構築時の作業領域量を大幅に減らすことができる．

また，Induced Sorting を用いて，接尾辞配列を構築せずに BWT を直接構築できる方法も知られている[26]．

3.3.4 BWT の逐次構築

ここでは，入力文字列から SA を介さず，BWT を逐次的にかつ直接構築する方法を紹介する．

文字列 T を $T[1], T[2], \cdots$ と前から順に処理し BWT を逐次的に更新して構

Algorithm 1 文字列 $T[0,n)$ の BWT を逐次的に構築する

```
function BWTIncremental(T[0, n))
    p←0
    for i←1, n do
        c←T[n−i]
        insert(T_B, p, c)
        p←rank_c(T_B, p)+C[c]
        C[c]←C[c]+1
    end for
    return T_B
end function
```

築した場合，計算量を抑えることは難しい．なぜなら，末尾に文字が追加されるたびに，すべての接尾辞の順序が変わる可能性があるためである．例えば $T=$“$bbb\$$” の接尾辞配列は $\{3,2,1,0\}$ であるが，$T=$“$bbbc\$$” となった場合，すべての接尾辞の順序が逆転し，$\{0,1,2,3,4\}$ となる．

一方，$T[n-1], T[n-2], \cdots$ と文字を後ろから順に追加し，それに対応する BWT を順次構築するのは容易である．

ここでは $S_{n-k+1}, S_{n-k+2}, \cdots, S_{n-1}$ までの接尾辞がソートされ，これに対応する BWT 後の配列 $T_B[0,k)$ が得られていたとし，次の文字 $c=T[n-k]$ が与えられ，接尾辞 S_{n-k} を先ほどの接尾辞配列に挿入する場合を考える．このとき，S_{n-k} を追加したとしても，他の接尾辞間の順序は変わらないことに注意されたい．よって S_{n-k} の $S_{n-k+1}, S_{n-k+2}, \cdots, S_n$ 中の順位 r を求め，$T_B[r]$ の位置に c を挿入する操作が実現できれば，S_{n-k}, $S_{n-k+1} \cdots S_{n-1}$ までの接尾辞に対する BWT が得られる．

さて，p を直前の S_{n-k+1} が挿入された位置とする．S_{n-k} より小さい接尾辞の数は，$T_B[0,p)$ 中に含まれる c の数，および c より小さい文字で始まる接尾辞の数の合計である．よって，S_{n-k} の順位は $\mathbf{rank}_c(T_B,p)+C[c]$ として求められる．

BWT を短い接尾辞から長い接尾辞を順に挿入して求める擬似コードを Algorithm 1 に示す．**insert**(T_B,p,c) は $T_B[p]$ と $T_B[p+1]$ の間に文字 c を挿入

$T = cadabra\$$ に対する BWT

i	T_B	接尾辞
0	a	\$
1	r	$a\$$
2	d	$abra\$$
3	c	$adabra\$$
4	a	$bra\$$
5	\$	$cadabra\$$
6	a	$dabra\$$
7	b	$ra\$$

(2) $p = \mathbf{rank}_a(B, 5) + C[a] = 3$
を求め，
$acadabra\$$ を
3 番目の位置に挿入
($T_B[3]$ の位置に \$ を挿入)

(1) $T_B[5]$ の \$ を a に置き換える

$T = acadabra\$$ に対する BWT

i	T_B	接尾辞
0	a	\$
1	r	$a\$$
2	d	$abra\$$
3	**\$**	***acadabra\$***
4	c	$adabra\$$
5	a	$bra\$$
6	***a***	$cadabra\$$
7	a	$dabra\$$
8	b	$ra\$$

図 3.5 BWT の逐次構築の例

する操作である．

図 3.5 に T=“$cadabra\$$” までの BWT が与えられたときに，この先頭に “a” が追加され，T'=“$acadabra\$$” となったときの BWT の更新の例を示す．初めの状態では p=5 であるので，**insert**(T_B, 5, “a”) とした後に，$p = \mathbf{rank}_a(T_B, 5) + C[a] = 3$ であるので，接尾辞としては “$acadabra\$$” が 3 の位置に挿入され，$T_B[3]$ に “\$” を挿入する．

この構築アルゴリズムは非常に簡単である．しかし，実用上困難なのは挿入操作をサポートしつつ，$\mathbf{rank}_c(T_B, p)$ を効率的に計算することであり，接尾辞配列を介した構築方法と比べて，まだ遅い．

また，この逐次更新では文字列を後ろから順に処理しているが，接尾辞の代わりに接頭辞を扱っているのだと考えれば，文字列を前から順に処理できる．文字列探索において，接尾辞である必要性はなく，接頭辞であってもこれまでの議論をまったく同様に適用できる．ストリームデータといった文字が次々とやってくる場合には，接頭辞配列，および接頭辞配列に対する BWT を考えればよい．

3.4 BWT の性質と復元

最後に，BWT 配列の重要な性質とそれを利用した復元をするための具体的な手順を述べる．3.2 節では各文字に番号をふることで T_B と T_F の対応付けをしたが，本節ではその対応付けを文字列上の操作で表現し，BWT の重要な性質をまとめる．はじめに配列をいくつか定義する．$ISA[0,n)$ を $ISA[SA[i]]=i$ となる配列だとする．ISA は SA の逆関数であり，接尾辞 S_i の接尾辞中の順位を返す関数だと考えらえる．この配列を利用し $T_B[i]$ に対応する T_F 中の位置は，次の位置にある接尾辞の順位であるので $ISA[SA[i]+1]$ と表現できる．また前と同様に $C[0,\sigma)$ を $C[c]$ が $c{\in}\Sigma$ より小さい文字の T 中の出現回数であるような配列とする．

このとき，次の定理が成り立つ．

定理 3.8 $T_B[i]=c$ のとき，

$$ISA[SA[i]-1] = \mathbf{rank}_c(T_B, i)+C[c] \tag{3.2}$$

□

［証明］ $T_B[i]=c$ のとき，$p=\mathbf{rank}_c(T_B, i)$ は $T_B[0,i)$ 中に出現した c の数を表し，$T_B[i]$ の c が c の中で何番めに出現していたかを表す．また，T_F は辞書式順序の 1 文字めなので，文字 c が開始する位置は $C[c]$ である．よって，p 番目の c が出現する位置は，$C[c]+p$ となる． ∎

Algorithm 2 文字列 T の BWT 後の文字列 T_B から，T を復元する

```
function BWTInverse(T_B[0, n), p)
    for i←0, σ−1 do
        C[i]←0
    end for
    for i←0, n−1 do
        c←T_B[i]
        C[c]←C[c]+1
    end for
    sum←0
    for i←0, σ−1 do
        cur←C[c]
        C[c]←sum
        sum←sum+cur
    end for
    for i←0, n−1 do
        c←T_B[i]
        ψ[C[c]]←i
        C[c]←C[c]+1
    end for
    for i←0, n−1 do
        T[i]←T_B[p]
        p←ψ[p]
    end for
    return T
end function
```

このBWT中の位置から，それに対応する T_F 中の位置への変換をLF-mappingと呼ぶ．このLF-mappingは全文索引や統計量の計算にも用いられる．

LF-mappingと逆の変換も次のように求められる．

定理 3.9 $T_F[i]{=}c$ のとき，

$$ISA[SA[i]+1] = \mathbf{select}_c(T, i-C[c]) \tag{3.3}$$

□

この関係は 7.5.1 節で ψ 配列として再度扱う．

BWT では接尾辞の辞書式順序でソートされており位置情報は失われているが，上の 2 つの関係を利用すれば T のもとの位置を前後に移動できる．これらの情報を組み合わせて BWT からもとの文字列を復元する擬似コードを Algorithm 2 に示す．はじめに，T の最後の文字 $T[n-1]=\$$ が T_B 中で現れる位置を p とする．この位置は BWT 時に一緒に保存しておく．次に T_B を調べ $C[0,\sigma)$ を作り次に (3.3) の式で表される T_F 中の位置から，対応する T_B 中の位置への変換を保持した配列である $\psi[0,n)$ を作る．あとは，$T_B[i]=\$$ の場所から順に，$\psi[i]$ を利用して文字列を 1 文字ずつたどり前から順に復元することができる．

4

簡潔データ構造

本章では，大規模な文字列データ解析において重要な役割を果たす簡潔データ構造を紹介する．簡潔データ構造は，操作が高速でありながら作業領域量が小さいという２つの性質を兼ね備えたデータ構造であり，データ圧縮と索引の技術が組み合わさったものである．なかでも特に重要となる完備辞書，簡潔木について解説する．

4.1　圧縮と索引の融合

これまで，計算機の演算処理性能はムーアの法則にしたがって向上し続ける一方，記憶装置のアクセス速度の向上速度はそれに追いつかず，演算速度と記憶装置のアクセス速度の差は広がっていった．こうしたトレンドから，近年のデータ処理においては計算回数を減らすよりも，いかに上位の高速な記憶階層で計算処理を実現できるかが重要となっている．

もう 1 つデータ処理の方法の観点から見た場合，多くの解析の場面では同じデータに対して繰り返し，さまざまな解析を行うという特徴がある．情報検索はその代表例であり，同じデータに対し，さまざまな検索操作を繰り返し適用する．この場合，データをあらかじめ分析しておき，索引と呼ばれる補助データ構造を構築しておくことで，毎回データ全体を見なくても，索引を利用して解析に必要なデータのみを参照できるようになる．最初 1 回の索引構築処理のコストは必要だが，索引さえできれば，毎回の解析操作を高速化できる．

簡潔データ構造は，データ構造を最適に近いサイズで保存しつつ，索引を利用して高速な操作を実現したデータ構造である．ここで最適というのは，データ構造がとりうる種類数が L のとき，これ以上小さくできないサイズの下限である $\lg L$ ビットに近いサイズであることを意味する．

データを小さくする技術としては，第 6 章で述べるデータ圧縮がよく知られている．データ圧縮の目標はデータを小さくするのみであり，データを扱う際は一度復元操作を行いデータをもとに戻す必要があるが，簡潔データ構造は復元操作をせず，さまざまな操作をそのまま行うことができる．ただ，近年データ圧縮で簡潔データ構造が多く使われることもあれば，簡潔データ構造でデータ圧縮が使われることもあり，この 2 つの技術分野の境界は曖昧になってきている．

簡潔データ構造はこうした計算機環境を背景に，大規模データを扱ううえでは重要なデータ構造となっている．

Pos	0	1	2	3	4	5	6	7	8	9	10
B	0	1	1	0	1	1	0	0	1	0	0

$\mathbf{rank}_1(B, 6) = 4$
$B[0, 6)$ 中の “1” の出現回数

Pos	0	1	2	3	4	5	6	7	8	9	10
B	**0**	1	1	**0**	1	1	**0**	**0**	1	0	0

$\mathbf{select}_0(B, 3) = 7$
B 中の(3+1)番目の “0” の出現位置

図 4.1　完備辞書の例

4.2　完備辞書

簡潔データ構造の中で最も基本的かつ重要なのが，ビット列に対する簡潔データ構造である完備辞書(Fully indexable dictionary，FID)である．

定義 4.1(完備辞書)　ビット列 $B[0,n), B[i]\in\{0,1\}$ に対して，次の操作を備えたデータ構造を完備辞書と呼ぶ．

- $\mathbf{access}(B,i)$: $B[i]$ を返す．
- $\mathbf{rank}_b(B,i)$: $B[0,i)$ 中の $b\in\{0,1\}$ の数を返す．
- $\mathbf{select}_b(B,i)$: B 中で先頭からみて $i+1$ 番目に出現した $b\in\{0,1\}$ の位置を返す．

□

図 4.1 に完備辞書の操作の例を挙げる．例えば，$\mathbf{access}(B,0)=0$, $\mathbf{access}(B,5)=1$ であり，$\mathbf{rank}_1(B,6)$ は $B[0,6)$ 中に 1 が 4 回出現しているので，$\mathbf{rank}_1(B,6)=4$ である．同様に，$\mathbf{select}_0(B,3)$ は，B の先頭から (3+1) 番めの 0 の出現位置は 7 であるので，$\mathbf{select}_0(B,3)=7$ である．

また，簡単にわかるように，$\mathbf{rank}_0(B,i)+\mathbf{rank}_1(B,i)=i$ が成り立つので，$\mathbf{rank}_0(B,i)=i-\mathbf{rank}_1(B,i)$ の関係が成り立つ．先ほどの例では，$\mathbf{rank}_0(B,6)=6-\mathbf{rank}_1(B,6)=6-4=2$ となる．

次に，**rank** と **select** の関係について考えてみる．**select**(B,i) は **rank**$(B,p)=i$, **rank**$(B,p+1)=i+1$ となる p の値であり，**rank** を定数時間で求められれば **select** は二分探索で $O(\log n)$ 時間で求められる．また，その逆に **select** を定数時間で求められれば **rank** は二分探索で $O(\log m)$ 時間で求められる，ただし m は B 中の 1 の数である．

この完備辞書をいかに実現するかについては後で詳しく述べるとして，完備辞書を利用してどのようなことが実現できるかについて，いくつか例を紹介しよう．

4.2.1　完備辞書の利用例

完備辞書は応用が広く，ほとんどの簡潔データ構造で内部表現として利用される．以下に代表的な使用例を見てみよう．

集合の表現

集合 $U=\{0,1,\cdots,n-1\}$ 中の部分集合 V は，$x\in V$ ならば $B[x]=1$，そうでなければ $B[x]=0$ としたビット列 B で表現できる．このとき，x が含まれているかどうかは **access**(B,x)，$i+1$ 番めに小さい V 中の要素は $\mathbf{select}_1(B,i)$，i より小さい V 中の要素の数は **rank**(B,i) として求めることができる．この他にも，x 未満の要素で V に含まれる最大の要素は，$\mathbf{select}_1(\mathbf{rank}_1(B,x)-1)$ として求められる．また，s 以上，e 未満の要素が V 中に何個含まれているかは，$\mathbf{rank}_1(B,e)-\mathbf{rank}_1(B,s)$ として求められる．

非負整数列の表現

正整数列 $P[0,m)$, $P[i]>0$ も完備辞書で表すことができる．ここで，2.5.1 節で紹介した単進符号を思い出してみよう．単進符号では正整数 $v>0$ を $v-1$ 個 0 を並べた後に 1 を 1 つだけ後続させたビット列で表される．

正整数列 $P[0,m)$ が与えられたとき，この各要素 $P[i]$ を単進符号で表したものを順につなげて得られたビット配列を $B[0,n)$ とする．このとき，ビット列の長さは $n=\sum_{i=0}^{m-1} P[i]$ であり，m 個の 1 からなる．

各 $P[i]$ の要素は，i 番めの 1 と $i+1$ 番めの 1 の間にある 0 の個数 +1 と一

Pos	0	1	2	3	4
P	4	1	1	2	3
Pの単進符号	0001	1	1	01	001

Pos	0	1	2	3	4	5	6	7	8	9	10
B	0	0	0	1	1	1	0	1	0	0	1

$P[0]+P[1]+P[2]$
$=\mathbf{select}_1(B, 2)-\mathbf{select}_1(B, -1)$
$=5-(-1)$
$=6$

$P[4]$
$=\mathbf{select}_1(B, 4)-\mathbf{select}_1(B, 3)$
$=10-7$
$=3$

図 4.2 完備辞書を用いて正整数列 $P[0,5)$ を表現した例

致し，$P[i]=\mathbf{select}_1(B,i)-\mathbf{select}_1(B,i-1)$ として求められる．ただし，$\mathbf{select}_1(B,-1)=-1$ と定義する．

また，P 中の連続する部分列の合計 $\sum_{i=s}^{e} P[i]$ は，s 番めの 1 と $e+1$ 番めの 1 の間の長さ +1 に一致し，次のように求められる．

$$\sum_{i=s}^{e} P[i] = \mathbf{select}_1(B,e)-\mathbf{select}_1(B,s-1) \tag{4.1}$$

図 4.2 に，例をあげる．$P[4]$ の場合は，$\mathbf{select}_1(B,4)-\mathbf{select}_1(B,3)=3$ として求められ，$\sum_{i=0}^{2} P[i]=\mathbf{select}_1(B,2)-\mathbf{select}_1(B,-1)=6$ として求められる．

また，任意の整数 x, $0\leq x\leq\sum_i P[i]$ について，

$$\sum_{i=0}^{k-1} P[i] \leq x < \sum_{i=0}^{k} P[i] \tag{4.2}$$

となるような k を

$$k = \mathbf{rank}_1(B,x) \tag{4.3}$$

として求められる．正整数列の完備辞書による表現は，可変長の符号やポインタを取り扱う際に利用される．

4.2.2 完備辞書の実現

完備辞書をどのように実現するかについて説明する．ビット列はそのまま格納すれば n ビットで格納でき，そのうえで **access** 操作を定数時間で実現できることは自明である．しかし，**rank**, **select** 操作を同じように定数時間，またはそれに近い計算時間で求めるためには工夫が必要である．

以降では $\mathbf{rank}_0$ は $\mathbf{rank}_1$ から求まるので，$\mathbf{rank}_1$，$\mathbf{select}_1$，$\mathbf{select}_0$ についてどのように実現するかを考える．

ビット列の長さを n とする．また，B 中の 1 の数を B の重みと呼ぶこととし，m で表す．

長さ n，重み m のビット列の 0 次経験エントロピーは，次のように求められる．

$$\begin{aligned} nH_0(B) &= -m\lg(m/n) - (n-m)\lg(1-m/n) \\ &\le m\lg(n/m) + m\lg\exp(1) \end{aligned} \tag{4.4}$$

$\lg\exp(1)$ はだいたい 1.44 である．m 個の 1 の位置を記録する場合，各相対位置の大きさは n/m 程度なので，それらを二進表現で記録した場合が $m\lg(n/m)$ であり，それに近い値がビット列を保存するのに必要なサイズである．

ビット列の 0 次経験エントロピーは，$m/n{=}1/2$ のときに最大値をとり，$m/n{=}0$, $m/n{=}1$ に近づくにつれ急速に小さくなっていく．$m/n{=}1/2$ の場合は $nH_0{=}n$ なので，ビット列をそのまま格納した場合が最適となる．それに対し，$m/n{=}0$ ($m/n{=}1$) に近づいていった場合は，nH_0 は $m\lg n$ ($(n-m)\lg n$) に近づくため，1 の出現位置 (0 の出現位置) をそのまま二進表現で格納した場合が最適となる．また，このような自明な表現で最適を達成する場合でも，**rank**, **select** 操作を高速に実現しつつ，作業領域量を小さくするには工夫が必要である．

ここでは，ビット列の重みが半分に近い場合 (密な場合)，重みが極端に小さい場合 (疎な場合)，0 と 1 の出現場所に偏りがある場合，それぞれの場合について，完備辞書をどのように実現するかを順に説明していく．

4.2.3　密な場合

密な場合，つまり $m/n=1/2$ に近い場合，0 次経験エントロピーはほぼ n であり，ビット列をそのまま格納しても 0 次経験エントロピーを達成できる．

それでは，**rank**，**select** をいかに実現するかを考えてみよう．はじめに，すべての位置の **rank** の結果をあらかじめ求めて，それを記録して参照する場合を考えてみよう．このとき，**rank** 操作は表を引くだけなので定数時間で実現できるが，作業領域量は 1 つの位置ごとに $\lg n$ ビット必要であり，位置の数は n 個あるので $n \lg n$ ビットになってしまう．目標は **rank** 操作を定数時間で実現しつつ，作業領域量が $n+\mathrm{o}(n)$ ビットを達成する方法であり，$n \lg n$ ビットはまだ大きい．

そこで，ビット列を適当な大きさのブロックに分割する．ビット列 B を長さ r ビットごとのブロックに分割し，表 $R[0, n/r)$ に，各ブロックの先頭の **rank** の結果を $R[i]=\mathbf{rank}(B, ir)$ と格納する．そして，長さ r ビットのブロックのすべてのビットパターンの場合の **rank** の結果を計算して，$P[x]$ に正整数 x を二進表現したときの **rank** の結果を格納する．この表を用いて長さがちょうど r ビットであるビット列中の 1 の数は表引きで求められる．また長さが r 未満のビット列 x の 1 の数を求めるには，残りの $r-s$ 個は 0 を埋めたビット列 x' を作り，$P[x']$ を参照すれば，x 中の 1 の数を調べることができる．

表 P を用いて長さが r 以下のビット列 x 中の 1 の数を数える操作を $p(x)$ とする．すると，**rank** 操作は $\mathbf{rank}_1(B, i)=R[i/r]+p(B[i//r, i))$ として表引き 2 回で求めることができるので，定数時間で求めることができる．

必要な作業領域量はどのくらいになっているのかを見てみよう．表 R の各要素は $\lg n$ ビットで表現できるため，表 R を格納するのに $(n/r)\cdot\lg n$ ビット必要である．また，表 P の各要素のサイズは $\lg r$ ビットであるので，表 P は $2^r \lg r$ ビットである．よって，ビット列本体以外のデータ構造のサイズは，$n/r \lg n+2^r \lg r$ ビットである．

目標は，ビット列本体以外のデータ構造のサイズを $\mathrm{o}(n)$ ビットに抑えることであった．しかし，先ほどの方法では，残念ながら r をどのような値にしても $\mathrm{o}(n)$ ビットで表すことはできない．なぜなら，r が $\lg n$ より小さい場合，

表 R のサイズが n より大きくなってしまうし，逆に r が $\lg n$ より大きい場合は，表 P のサイズが n より大きくなってしまうからである．

1度でダメなら2度ということで，ブロックをさらに再帰的に分割してみよう．配列 B を長さ l ビットのブロックに分割し，これらを大ブロックと呼ぶことにする．次に各大ブロックを長さ s ビットのブロックに再度分割し，これらを小ブロックと呼ぶことにする．表 $L[0,n/l)$ に，各大ブロックの先頭の **rank** の結果を格納し，$L[i]=\mathbf{rank}_1(B,il)$ とする．また，表 $S[0,n/s)$ に，各小ブロックの先頭での **rank** の結果から，大ブロックの先頭までの **rank** の差を格納し，$S[i]=\mathbf{rank}_1(B,is)-L[is/l]$ とする．最後に長さ s のブロック内でのすべてのビットパターンの **rank** の結果をあらかじめ求めておき，表 $P[0,2^s)$ に格納する．

この場合，**rank** 操作は $\mathbf{rank}_1(B,i)=L[i/l]+S[i/s]+P[B[i//s,i)]$ として表引き3回で求めることができるので，定数時間で求めることができる．

表 L の各要素は $\lg n$ ビットで格納できるので，L のサイズは $(n/l)\cdot\lg n$ ビットである．表 S の各要素は $\lg l$ ビットで格納できるので（$S[i]$ の値は最大で $l-1$ までしかとらないため），S のサイズは $(n/s)\cdot\lg l$ ビットである．最後に表 P の各要素は $\lg s$ ビットなので，表 P は $2^s\lg s$ ビットで格納できる．よって，補助データ構造の合計サイズは $(n/l)\cdot\lg n+(n/s)\cdot\lg l+2^s\lg s$ ビットとなる．ここで $l=\lg^2 n$，$s=(\lg n)/2$ とすれば，合計サイズを $\mathrm{o}(n)$ ビットに抑えられる．

図4.3に，$B[0,n)$ に対し，$n=256$ の場合の例を挙げる．$\lg n=8$ であるので，$l=\lg^2 n=64$，$s=(\lg n)/2=4$ である．そして，$\mathbf{rank}_1(B,114)$ を求める場合には，$L[114/l]=L[1]$，$S[114/s]=S[28]$ を参照する．最後に，$S[28]$ の内部では $114 \bmod 4=2$ であるので，28番目の小ブロック内のビット列 1010_2 に対し，最初の2ビット中の1の数を数えるために，1100_2 でマスクをとり $1010_2\&1100_2=1000_2$ となり，この中の1の出現数を $P[1000_2]$ で求める．これらより，**rank** の値は $\mathbf{rank}_1(B,114)=L[1]+S[28]+P[1000_2]$ で求められる．以上より，**rank** 操作を定数時間で実現しつつ，作業領域量が $n+\mathrm{o}(n)$ ビットであるようなデータ構造が作れることを示せた．

同じように **select** 操作の場合も定数時間かつ $n+\mathrm{o}(n)$ ビットで構築できる

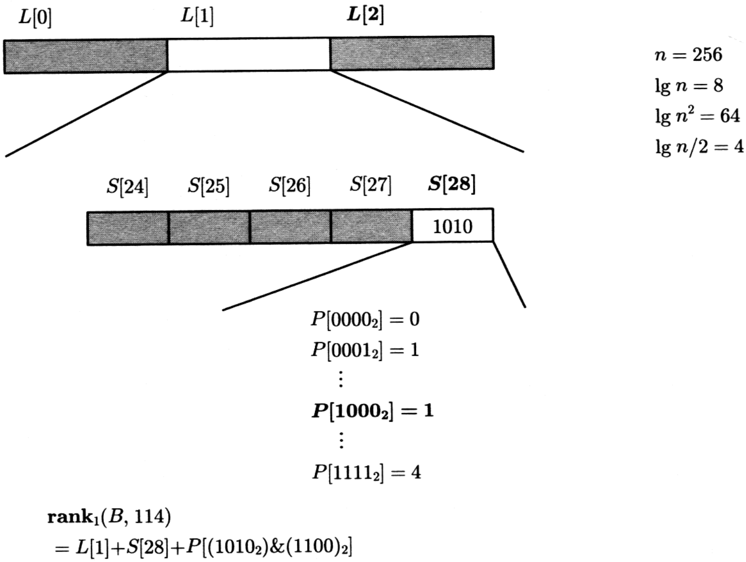


図 4.3 密なビット列に対する完備辞書の表現方法

が，**rank** 操作よりも複雑なデータ構造が必要である．そのため多くの場合は，**select** は **rank** を利用して二分探索を行って求める．**rank** 操作は定数時間なので，$O(\lg n)$ 時間で求めることができる．また，表 L，S を直接利用すれば，漸近的な性能は変わらないが，実用的には次のようにしてさらに速くできる．はじめに，表 L を利用して二分探索を行い，$L[p]\geq i$ となるような最小の p を探す．次にこの大ブロック内の範囲で二分探索を行い，$L[p]+S[q]\geq i$ となるような最小の q を探す．最後に，ブロック内での **select** は，表引きによって定数時間で求められる．

4.2.4 疎な場合

次にビット列が疎な場合を考えてみる．疎な場合は 1 の数が少ないので，1 の出現位置をそのまま二進表現で格納しても全体のサイズは抑えられる．具体的には，長さが n，重みが m であるビット列が与えられたとき，$P[0,m)$，$P[i]=\textbf{select}_1(P,i)$ とした配列をそのまま二進表現で格納した場合，サイズは

$m\lg n$ ビットである．この場合，$\mathbf{select}_1(P,i)$ は $P[i]$ そのものであり，定数時間で計算できる．また，$\mathbf{select}_0(P,i)$, $\mathbf{rank}_1(P,i)$ は $\mathbf{select}_1$ を用いた二分探索で求められるので $\mathrm{O}(\log m)$ 時間で求めることができる．

さて，この $m\lg n$ ビットは最適からどれくらい離れているだろうか．先ほど調べたように，このビット列の 0 次経験エントロピーは $m\lg(n/m)+m\lg\exp(1)$ で抑えられる．つまり，この表現はだいたい $m(\lg m-\lg\exp(1))$ 分，0 次経験エントロピーより大きいことになる．例えば，$n=2^{32}$, $m=2^{24}$ の場合，$m\lg n=2^{24}\cdot 32$，$m\lg(n/m)+m\lg\exp(1)=2^{24}(8+1.44)$ であり，後者は前者の 1/4 程度である．以降では，最適なサイズを達成する方法を紹介する．

はじめに $P[0,m), P[i]=\mathbf{select}_1(B,i)$ とする．次に，$p=\lfloor\lg(n/m)\rfloor$ とし，$P[i]$ の下位 p ビットをそのまま $L[i]$ に格納し，残りの上位ビットを $H[i]=P[i]/2^p$ と格納する．この符号法はパラメータを 2^p としたライス符号で P を表現した場合と同じであり，$P[i]=2^pH[i]+L[i]$ である．

整数列 $H[0,m)$ は広義単調増加列であることから，$D[i]=H[i]-H[i-1]$ であるような配列 $D[0,m)$ を構成した場合，D は非負整数列である，ただし $H[-1]=0$ とする．この D を利用して，$H[i]=\sum_{k=0}^{i} D[i]$ として表すことができる．また D は非負整数列なので，4.2.1 節で紹介したように長さ $2m$ ビットのビット列 E で表すことができる．もとのビット列 B が長さ n，重み m である疎なビット列であったのに対し，新しく作った E は長さ $2m$，重み m の密なビット列であることに注意する．このとき，E は密なビット列に対する完備辞書を利用し $2m+\mathrm{o}(m)$ ビットで **rank**, **select** 操作を備えることができる．

全体の作業領域量を見てみよう．下位ビット列 L は $m\lg(n/m)$ ビットであり，H は $2m$ ビット，そして H に対する完備辞書のオーバーヘッドの分が $\mathrm{o}(m)$ ビットである．よって，全体の作業領域量は，$m\lg(n/m)+2m+\mathrm{o}(m)$ ビットである．この値は 0 次経験エントロピーに近い値になっている．

この L と E を利用して **select**, **rank** をどのように実現していくかを見てみる．まず，$\mathbf{select}_1$ は，

$$\mathbf{select}_1(B,i) = 2^p(\mathbf{select}_1(E,i)-i)+L[i] \tag{4.5}$$

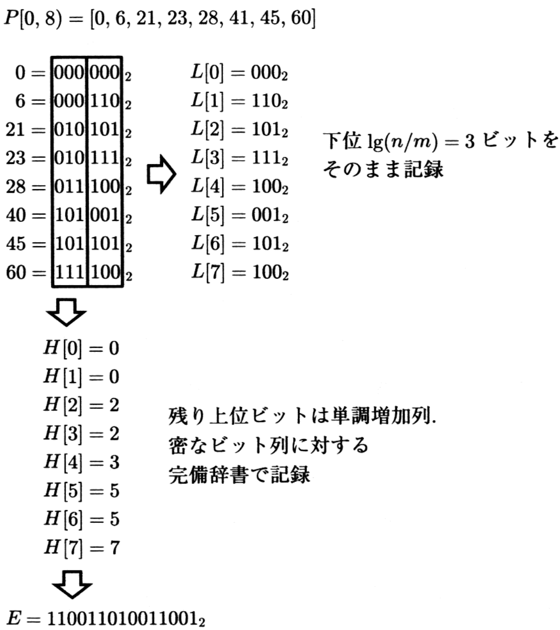


図 4.4 疎なビット列に対する完備辞書の表現方法

として求められる.

図 4.4 に n=64, m=8 の場合の例を挙げる. 下位 $\lg(n/m)$=3 ビットが L にそのまま記録され, 残りの上位ビットが E として表現される. 例えば, $\mathbf{select}_1(B,6)$ は, $2^3 \cdot (\mathbf{select}_1(E,6)-6)+L[6]=8\cdot5+5=45$ で求められる.

次に $\mathbf{rank}_1(B,i)$ であるが, E で 0 が 1 回出現するたびに, もとの値が 2^p だけ増加しているということに注意すると, i 番目の位置は E 中では $(i/2^p)+1$ 番めの 0 以降に対応しているのがわかる. この位置は $t=\mathbf{select}_0(E,i/2^p)+1$ として求めることができる. そして, $H[t]$ 以降で次に $H[t]$=1 である間, $L[t]$ が $i/2^p$ より小さいかどうかを調べる. この操作で調べなければならない最大の個数は 2^p=$\lg(n/m)$ 個であるため(それ以上になると値が 2^p だけ増え, $H[t]$=0 となる), $\mathbf{rank}_1$ の最悪計算量は $O(\log(n/m))$ である. しかし, もとのビット列中の 1 の位置が一様分布している場合には, 連続する 1 の数の期待値は定数である. よって, $\mathbf{rank}_1$ の計算量の期待値は定数時間である. ま

た，**access**(B, i) も $\mathbf{rank}_1$ と同様にして求めることができる．この場合，L 側を調べる際には，$L[u]{=}i/2^p$ であるかどうかを調べればよい．

4.2.5 出現場所が偏っている場合

最後に扱うビット列は，1 の出現場所が偏って，ある部分では 0 が多く，他の部分では 1 が多いような場合である．例えば，長さが $2n$ のビット列で，最初の n 個が 1 のみ，最後の n 個が 0 のみである場合を考えてみる．この場合は 1 の出現割合だけからみれば密なビット列であるが，そのまま $2n$ ビットで格納するよりはるかに小さく格納できるはずである．このような偏りのあるビット列の効率的な格納は重要であり，これから紹介する全文索引や，多くのデータ構造を扱う際に重要になってくる．

このような偏りのあるビット列を効率的に格納したまま，**access**, **rank**, **select** 操作を高速に実現する方法[28]を紹介する．この手法は提案者らの頭文字から RRR と呼ばれている．

はじめに，ビット列のブロック b に対し，クラスとオフセットを導入する．ブロック b 中の 1 の数を，そのブロックのクラスと呼び，$c(b)$ で表す．例えば，$b{=}$“010010” の場合，$c(b){=}2$ である．ブロック b のオフセット $o(b)$ を，同じクラスに割り当てられるブロックの中で，b より辞書式順序で小さいブロックの数とする．先ほどのブロック b が長さ 6，1 の数が 2 個であるブロックの中で何番目に小さいかを調べると，そのようなブロックは小さい順に “000011”，“000101”，“000110”，“001001”，“001010”，“001100”，“010001”，“010010”，… であり，“010010” は 8 番目に小さいので，$o(b){=}7$ である．ビット列はクラスとオフセット情報だけから復元できることに注意する．例えば，$c(b){=}2, o(b){=}7$ から，$b{=}$“010010” が一意に定まる．クラスとオフセットの情報からもとのビット列をどのように効率的に復元するかはこの節の後半で説明する．

長さ n，重みが m であるビット列 $B[0, n)$ を長さ $t{=}(\lg n)/2$ ごとに分割し $b[0, n/t)$, $b[i]{=}B[it, (i{+}1)t)$ のように分割する．次に，各ブロックについて，それらのクラスとオフセットを計算し，それぞれ $C[0, n/t)$ と $O[0, n/t)$ に次のように格納する $c[i]{=}c(b[i])$, $O[i]{=}o(b[i])$.

各ブロックのクラス $c[i]$ は $0 \leq c[i] \leq t$ であるので，$\lg t$ ビットの固定長で格納し，任意のクラスは定数時間でアクセスができる．それに対しオフセットは，長さ t 中で $c[i]$ 個の 1 を含むブロックの種類数は $\binom{t}{c[i]}$ であり，$\lg \binom{t}{c[i]}$ ビットで格納できる．オフセットを記録するのに必要なビット長の合計は B の 0 次経験エントロピーで抑えられる．

$$\sum_i \lg \binom{t}{c[i]} \leq nH_0(B) \tag{4.6}$$

また，ブロックごとに 0 と 1 の数の偏りが大きい場合に，$\binom{t}{c[i]}$ の値は小さくなり全体サイズはさらに小さくでき，0 次経験エントロピーよりもはるかに小さく格納できる．

オフセットを格納するときに問題となるのは各オフセットの符号長は可変長であり，各符号がどこからはじまっているのかがわからないことである．このため任意のオフセット情報を定数時間で取得できない．これを解決するために追加のデータ構造を用意する．各ブロックを長さ $s = t \lg n$ ごとの大ブロックごとにまとめ，各大ブロックには，大ブロックの開始位置の **rank** の値と，オフセットを格納している配列の開始位置のポインタを格納する．また，大ブロック中での開始位置からの相対開始位置を格納するのは，先ほどの疎なビット列に対する完備辞書を利用すれば定数時間で求めることができる．よって，全体のサイズを $nH_0(B)$ ビットで格納し，**rank**, **access** 操作を定数時間で操作できることが示せた．また，**select** についても同様に高速に求めることができるが，それについては省略する．

最後にブロックからオフセットを求める，またはその逆を行う際に利用する擬似コードを Algorithm 3 に示す．$C[i][j]$ は長さが i で，重みが j である場合の数を記録する．この配列は前計算で求めておくとする．例えば $n=7$, $m=3$ の場合，$o(1000101)=\binom{6}{3}+\binom{2}{2}+\binom{0}{1}=20+1+0=21$ であり，$o(1110000)=\binom{6}{3}+\binom{5}{2}+\binom{4}{1}=20+10+4=34$ となる．逆にオフセットからもとのブロックを復元するには逆の操作を行えばよい．

Algorithm 3 重みが m であるビット列 B のオフセットを返す

```
function EnumCode(B, m)
    code←0
    for i←0, n do
        if B[i]=1 then
            code←code+C[n−i−1][m]
            m←m−1
        end if
    end for
    return code
end function
```

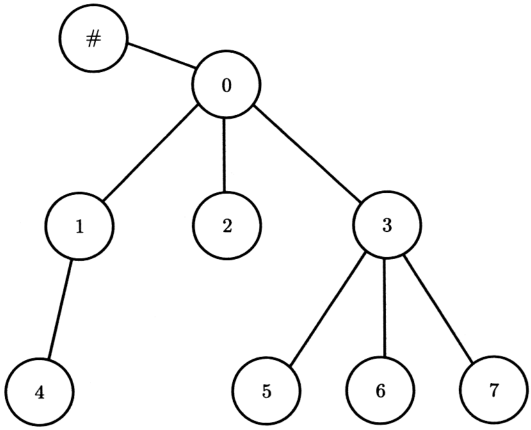
4.3 木に対する簡潔データ構造

次に木の簡潔データ構造である簡潔木を考えてみる．木とは閉路のない連結グラフのことであり，本書では節点の 1 つが根として指定された根付き木を考える．

現実世界の木とは逆に，データ構造での木の表現では根を一番上に書き，下に向かって枝分かれをしているように表す．木構造は節点と枝からなる．節点 x と y が枝 $e=(x,y)$ でつながっており，x が y より根に近い場合，x を y の親と呼び，y を x の子と呼ぶ．根を除きすべての節点は 1 つの親を持つ．同じ親を持つ節点を兄弟と呼び，子の数が 0 個の節点を葉と呼ぶ．以下では兄弟間に順序が付けられている順序木を扱う．

一般的な順序木の実現では各節点ごとに親と最初の子，そして次の兄弟の 3 つの節点を指し示すポインタを保持する．木が n 個の節点からなる場合，各ポインタは $\lg n$ ビット必要なので，各節点ごとに $3\lg n$ ビット必要となる．例えば $\lg n=64$ の場合，1 つの節点あたり 192 ビット必要である．

それでは，このような順序木を表現するために最低限必要なビット数はいくつかを調べてみよう．はじめに，n 個の節点(木の定義より，$n-1$ 個の枝)からなる順序木が何通りあるのかを考えてみる．これはカタラン数と呼ばれる



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	0	1	0	0	0	1	0	1	1	0	0	0	1	1	1	1	1
0 ベース表現	0		1	2	3		4			5	6	7					
1 ベース表現		0				1		2	3				4	5	6	7	

図 4.5　LOUDS による木の簡潔表現

数 $C(n)$ に等しいことが知られており，n が十分大きくなると 4^n 程度になる．L 通りの場合を誤りなく区別して表すには $\lg L$ ビット必要である．よって，1 つの順序木を表すには $\lceil \lg C(n) \rceil = 2n$ ビットが必要であることがわかる．

順序木を $2n+o(n)$ ビットで表現し，さまざまな操作を実現する簡潔データ構造を簡潔木と呼ぶ．簡潔木を実現する方法はいくつか知られている．ここではその中でも LOUDS 表現について紹介する．

木を幅優先探索で辿り，各節点において子の数を単進符号で順に記録する．このとき，葉も子が 0 個の節点として扱う．このようにして作られたビット列は 1 の数が節点の数に等しく，0 の数が節点の数 -1 に等しくなっているので，ビット列の長さは $2n-1$ である．また，先頭に根の親であるような節

点を用意する．この節点は番人の役割を果たす．このような順序木の表現を LOUDS (Level-Ordered Unary Degree Sequence) 表現と呼ぶ．

LOUDS 表現において，各節点は子として出現したときの 0 と，自分の子の数を記録しているときの単進符号の末尾の 1 として 2 回表現されている．前者を 0 ベースの表現，後者を 1 ベースの表現と呼ぶ．

図 4.5 に LOUDS 表現の例を挙げる．図中の番号 3 がつけられた節点の場合，0 ベースの表現では根の 3 番めの子として表現されており ($B[4]=0$)，1 ベースの表現では 3 つの子を持つ節点 ($B[12]=1$) として表される．

LOUDS 表現では，ビット列上の **rank**, **select** 操作を介して，各節点の親，子，兄弟をたどることができる．$B[p]=0$ のとき，これに対応する 1 ベースの節点の位置は，$\mathbf{select}_1(B, \mathbf{rank}_0(B,p)+1)$ として求められる．これは子への移動に対応する．この逆の操作として，$B[p]=1$ に対応する 0 ベースの節点の位置は $\mathbf{select}_0(B, \mathbf{rank}_1(B,p))$ として求められる．これは親への移動に対応する．

まとめると，LOUDS 表現では，**rank** と **select** 操作のみを用いて，$2n+\mathrm{o}(n)$ ビットで木を自由にたどることが，完備辞書を利用して実現できる．

5

ウェーブレット木

ウェーブレット木は多様な目的に利用できるデータ構造である．もともとウェーブレット木は，文字列に対する **rank**, **select** 操作を効率的に実現するために提案された．その後，ウェーブレット木は驚くほど多くの操作を効率的に実現でき，また対象データは文字列だけでなく，2 次元グリッド情報，グラフ情報，転置索引，順序情報など多くの種類のデータを扱えることがわかってきた．本章では，ウェーブレット木について解説した後に，それが備えるさまざまな特性について説明する．

5.1 文字列上の操作の実現

ウェーブレット木について説明する前に，表 2.1 で紹介した文字列上の操作はウェーブレット木がない場合どのように実現できるかについて考えてみよう．まず，1 つめの $\mathbf{rank}_c(T,i)$ を求める問題を考えてみる．この問題は 4.2 節で扱った完備辞書の場合は $\sigma=2$ の場合であったが，$\sigma\geq 3$ に拡張された場合であると考えられる．

完備辞書のときと同じアプローチをとり，文字列を再帰的にブロックに分割し，各ブロックの先頭の **rank** の結果を足し合わせることで $\mathbf{rank}_c(T,i)$ を求めることはできる．しかし，この場合，アルファベットサイズが大きい場合に，各表のサイズがもとの文字列と比べて無視できないくらい大きくなってしまうという問題がある．アルファベットサイズは $n=\sigma$ になりうるため，こうした場合に表のサイズを抑えることは難しい．

次に，$T[s,e)$ 中の $(r+1)$ 番目に大きい値を返す $\mathbf{quantile}(T,s,e,r)$ を考えてみる．この場合，$w=e-s$ のとき，$T[s,e)$ をソートすれば $\mathrm{O}(w\lg w)$ 時間，また上位 $r+1$ 番目だけ求める手法を使えば $\mathrm{O}(w+r\lg r)$ 時間で求めることはできるが，w に依存しない計算量，かつ作業領域量がもとの文字列と同程度であるデータ構造の実現は困難であった．

ウェーブレット木はこれらを含め多くの文字列上の操作を同じ 1 つのデータ構造で $\mathrm{O}(\lg\sigma)$ 時間で実現し，必要な作業領域量はもとの文字列と同じという優れた性質を持つ．

5.2 ウェーブレット木の構築方法と特徴

この節ではウェーブレット木の構築方法とその特徴について紹介する．

文字列 $T[0,n)$, $T[i]\in\Sigma=\{0,1,\cdots,\sigma-1\}$ に対するウェーブレット木を考える．はじめにアルファベット木を説明する．アルファベット木は二分木である．アルファベット木の各葉が Σ 中の各アルファベットに対応し，各節点はその子孫の葉に対応するアルファベットの和集合に対応する．なお，葉のアルファベ

ットは左から小さい順に並べるとする．根はすべてのアルファベットに対応する．アルファベット木はハフマン木のときと同様に各文字が，根から葉へたどるパス上で左に行く場合は 0，右に行く場合は 1 としたビットをつなげた符号で表されていると考えることもできる．ウェーブレット木はアルファベット木と，各節点に付随するビット列から作られた完備辞書を組み合わせたデータ構造である．

このとき，ウェーブレット木は次のように構築される．アルファベット木の根に文字列 T 全体を対応させ，根に付随するビット列を B_* とする．次に，根の各文字 $c{=}T[i]$ について，もし文字 c が左の子の子孫に対応するならば，$B_*[i]{=}0$ とし，c を左の子に振り分ける．逆に c が右の子の子孫に対応するならば $B_*[i]{=}1$ とし，c を右の子に振り分ける．この操作をすべての文字に対して順に適用すると，B_* には各文字が左の子，右の子のどちらに振り分けられたかを表すビット情報が格納される．この振り分け操作をすべての文字が葉に振り分けられるまで，左の部分木，右の部分木それぞれに対して，再帰的に適用していく．

具体的な構築アルゴリズムを説明するために，ウェーブレット木の中のアルファベット木をたどる操作を定義する．節点 t が与えられたとき，その節点の左の子（0 側の子）を返す操作を **leftchild**(t) とし，右の子（1 側の子）を返す操作を **rightchild**(t) とする．また，**child**$(t,0)$=**leftchild**(t)，**child**$(t,1)$= **rightchild**(t) とする．節点 t に対し **isleaf**(t) は t が葉ならば `True`，葉でないならば `False` を返す関数とする．同様に，節点 t に対し **isroot**(t) は t が根ならば `True`，根でないならば `False` を返す関数とする．最後に $WT.root$ でウェーブレット木の根を返し，$WT.leaf(c)$ で文字 c に対応する葉を返すとする．

Algorithm 4 に，文字列 T に対するウェーブレット木の構築例を示す．図 5.1 に T=“*dacabdabcbdc*” の例を挙げる．根については，$T[i]\in\{a,b\}$ ならば，それらの文字は左の子の子孫に対応するので，$B_*[i]{=}0$ とし，$T[i]\in\{c,d\}$ であるならば，それらの文字は右の子の子孫に対応するので，$B_*[i]{=}1$ とする．この結果として $B_*{=}101001001011$ である．また左の子には T の中で a または b だけからなる文字列 “*aababb*” が対応し，右の子には c または d だけから

Algorithm 4　build(T): T のウェーブレット木 WT を構築する

```
function build(T[0, n))
    for i←0, n do
        c←T[i]
        t←WT.root
        d←1
        while isleaf(t)= False do
            b← c の d 番めのビット
            push_back(B_t, b)
            t←child(t, b)
            d←d+1
        end while
    end for
    return WT
end function
```

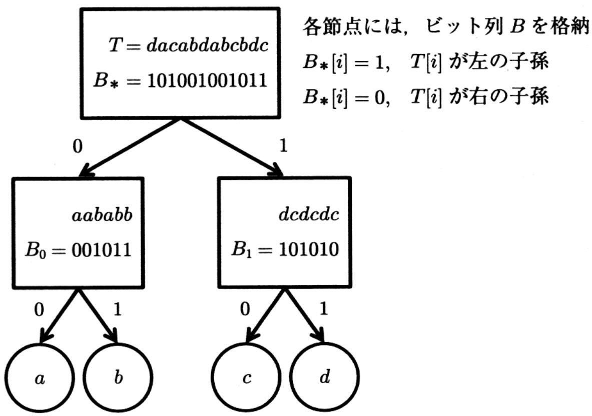


図 5.1　ウェーブレット木の例

なる文字列 “$dcdcdc$” が対応する．左の子に対しさらに再帰的に a の位置は左の子に対応するので 0，b の位置は右の子なので 1 が対応し，結果として B_0=001011 を得る．右の子についても同様に処理し，B_1=101010 が得られる．

こうしてすべての文字が割り振られた後，各節点に付随するビット列に対し

て完備辞書を構築し，各ビット列上で **access**, **rank**, **select** 操作が定数時間で求まるようにしておく．

ウェーブレット木の構築時間は，各文字ごとに $\lg\sigma$ 箇所の節点で操作をし，各節点での操作は定数時間で行うことができるので，合計で $O(n\lg\sigma)$ 時間である．また構築に必要なサイズはテキストとウェーブレットを別々に保存すれば $2n\lg\sigma$ ビットであるが，効率的な構築手法も提案されている[5]．

ウェーブレット木を格納するのに必要なサイズは各文字ごとに $\lg\sigma$ ビットであるので，$n\lg\sigma$ ビットである．このサイズは元の文字列 $T[0,n)$ を格納するのに必要であったサイズと同じである．ウェーブレット木は各文字の情報をビットごとに並び替えたものと考えられる．

アルファベット木の高さは $\lg\sigma$ である．なぜなら，葉の数が σ であるような完全二分木であるからである．よって，ウェーブレット木を根から葉へたどる操作は $O(\lg\sigma)$ 時間となる．また，同じ深さの節点に付随するビット列をつなげた長さは常に n となる．こうした特徴は後のより効率的なウェーブレット木の表現につながっていく．

操作例を説明する前に練習としてウェーブレット木を使ってもとの文字列を復元する方法についてみてみよう．各節点 v ごとに，ビット列をどこまで読み込んだかを示すポインタ p_v を用意し $p_v{=}0$ と初期化しておく．次に t を根に設定し，$b{:=}B_t[p_t]$ とし，$p_t{:=}p_t{+}1$ とする．そして，$t{:=}\mathbf{child}(t,b)$ と節点をたどる．葉に到達したらそれに対応する文字を出力する．この操作をすべての文字を復元するまで繰り返せばもとの文字列を 1 文字ずつ前から順に復元できる．復元には $O(n\log\sigma)$ 時間要する．

5.3 文字列操作の実現

ウェーブレット木はもともと文字列に対する **rank**, **select** 操作を効率的にサポートするために提案された[12]．その後，多くの操作を同様に効率的に実現できることがわかった．

ウェーブレット木は表 2.1 の文字列に対する操作をすべて高速にサポートし，必要な作業領域量はもとの文字列と同じかそれ以下しか必要としない．ア

Algorithm 5 **access**(T, i): 文字 $T[i]$ を復元する

```
function access(T, i)
    t←WT.root
    p←i
    while isleaf(t)= False do
        b←access(B_t, i)
        p←rank_b(B_t, p)
        t←child(t, b)
    end while
    return t が対応する文字
end function
```

ルファベットサイズが $n=\sigma$ のような場合や，アルファベットごとの頻度の偏りが大きい場合でも計算量は変わらないという優れたデータ構造である．

また，ウェーブレット木を実現するうえで必要なのは 4.2 節で扱った完備辞書のみである．データ構造の研究の多くは定数項が大きい，実装が困難であるといった問題があり実用的でない場合が多いが，ウェーブレット木はこれらの問題を克服した実用的なデータ構造である．

以下に各操作をウェーブレット木でどのように実現するかについてみてみる．

- **access**(T, i): 文字 $T[i]$ を復元する．

  Algorithm 5 に，擬似コードを示す．節点を指すポインタ t を根にセットし，節点に付随するビット列中の位置を示す変数 p を $p{:=}i$ とする．次に，$B_t[i]$ を調べることで i 番めの文字がどちらの子に振り分けられたのかを調べる．このビットの情報は **access**(B_t, i) で取得できる．仮に **access**$(B_t, i)=0$ であったとする．この場合，左の子に対応するはずであり左の子に振り分けられた文字の中で何番目であるかを調べる．何番めであるかは $B_t[0, i)$ 中に 0 が何回出現したかを調べることで求められ，この値は $\mathbf{rank}_0(B_t, i)$ で求めることができる．よって，$t{:=}\mathbf{leftchild}(t)$, $p{:=}\mathbf{rank}_0(B_t, p)$ とし，左の子から同じ操作を再帰的に繰り返す．同様に $B_t[i]=1$ であった場合も $t{:=}\mathbf{rightchild}(t)$, $p{:=}\mathbf{rank}_1(B_t, p)$ とし，右の

Algorithm 6 $\mathbf{rank}_c(T,i)$: $T[0,i)$ 中の文字 c の出現回数を返す

```
function rank(T, i, c)
    t←WT.root
    p←i
    d←1
    while isleaf(t)= False do
        b← c の d 番めのビット
        d←d+1
        p←rank_b(B_t, p)
        t←child(t, b)
    end while
    return p
end function
```

子から同じ操作を再帰的に繰り返す．この操作を繰り返し，到達した葉の文字を返す．各節点で必要な計算量は **access**, **rank** ともに定数時間であり，また木の深さは $\lg\sigma$ であるので，合計で $O(\log\sigma)$ 時間で $\mathbf{access}(T,i)$ は求めることができる．

- $\mathbf{rank}_c(T,i)$: $T[0,i)$ 中の文字 c の出現回数を返す．

  Algorithm 6 に擬似コードを示す．**rank** を組み合わせることで，任意の範囲内 $T[i,j)$ 中の文字 c の出現回数は $\mathbf{rank}_c(T,j)-\mathbf{rank}_c(T,i)$ として求められることに注意する．この **rank** は先ほどの **access** と同様に根から葉をたどる．はじめに節点を指すポインタ t を根に設定し，節点に付随するビット中の位置を指すポインタ p を $p:=i$ と設定する．次に，c がもし左の子に対応するのであれば B_t 中の 0 に対応するので，$B_t[0,p)$ 中に 0 が何回出現したかを調べればよい．この回数は $\mathbf{rank}_0(B_t,p)$ で求められるので，$p:=\mathbf{rank}_0(B_t,p)$, $t:=\mathbf{leftchild}(t)$ とする．同様に c がもし右の子に対応するのであれば $p:=\mathbf{rank}_1(B_t,p)$, $t:=\mathbf{rightchild}(t)$ とする．この操作を c に対応する葉にたどりつくまで繰り返し適用する．そしてそのときの p の値が $\mathbf{rank}_c(T,i)$ に対応する．計算量は $O(\log\sigma)$ 時間である．

Algorithm 7 $\mathbf{select}_c(T,i)$: T 中の $(i+1)$ 番めの c の出現位置を返す

```
function select(T, i, c)
    t←WT.leaf(c)
    p←i
    d←lg c
    while isroot(t)= False do
        b← c の d 番めのビット
        p←select_b(B_t, p)
        t←parent(t)
        d←d−1
    end while
    return p
end function
```

- $\mathbf{select}_c(T,i)$: T 中の $(i+1)$ 番めの c の出現位置を返す．
  Algorithm 7 に，擬似コードを示す．**access**, **rank** とは逆に葉から根に向かって順にたどっていく．節点を示すポインタ t を文字 c に対応する節点にセットし，ビット列中の位置を指すポインタ p を $p{:=}i$ とする．次に $u{:=}\mathbf{parent}(t)$ とし，t が u の左の子であったら，ビット b を 0 に，右の子なら b を 1 とする．もし左の子であったならば，c の中で p 番めの文字が節点 u のどの位置に出現していたのかを $p{:=}\mathbf{select}_b(B_u,p)$ で求める．そして $t{:=}u$ と更新する．この操作を根に到達するまで繰り返し，根での p を返す．計算量は $O(\log\sigma)$ 時間である．
- $\mathbf{quantile}(T,s,e,r)$: $T[s,e)$ 中の $(r+1)$ 番めに大きい値を返す．なお，同じ値が複数回出現する場合は，それらを重複して順位を数える．
  Algorithm 8 に，擬似コードを示す．この操作は例えば，$r{=}0$ ならば $T[s,e)$ 中の最大値，$r{=}e{-}s{-}1$ ならば $T[s,e)$ 中の最小値を返すことができることに注意する．はじめに，ビット列中の開始位置 st と終了位置 en を示すポインタを $st{:=}s$, $en{:=}e$, さらに現在何番めに大きい値を調べているかを示す変数 $remain$ を $remain{:=}r$，節点を示すポインタ t を根にセットする．次に，$B_t[s,e)$ 中の 0 の出現回数 zn を $zn{=}\mathbf{rank}_0(B_t,e)-$

Algorithm 8 **quantile**(T,s,e,r): $T[s,e)$ 中の $(r+1)$ 番めに大きい値を返す

```
function quantile(T, s, e, r)
    t←WT.Root
    st←s
    en←e
    remain←r
    while isleaf(t)= False do
        zn=rank_0(B_t, en)−rank_0(B_t, st)
        if remain≤zn then
            b←0
        else
            b←1
            remain←remain−zn
        end if
        st←rank_b(B_t, st)
        en←rank_b(B_t, en)
        t←child(t, b)
    end while
    return t
end function
```

$\mathbf{rank}_0(B_t,s)$ として求める．もし，$remain<zn$ であれば，$remain$ 番めに大きい文字は B_t 中の 0 に対応する側の子にあることがわかるので $st{:=}\mathbf{rank}_0(B_t,st)$, $en{:=}\mathbf{rank}_0(B_t,en)$ と更新し，$t{:=}$**leftchild**(t) とする．このとき，$remain$ はそのままである．そうではなく，$remain\geq zn$ であれば，$remain$ 番めに大きい文字は B_t 中の 1 に対応する側の子に含まれているので，$st{:=}\mathbf{rank}_1(B_t,st)$, $en{:=}\mathbf{rank}_1(B_t,en)$ と更新し，$t{:=}$**rightchild**(t) とする．また 0 側には zn 個出現していることがわかるので，1 側の子では $remain-zn$ 番めに大きい文字を調べればよい．よって，$remain{:=}ramain-zn$ と更新する．この操作を葉に到達するまで繰り返し，そのときの葉に対応する文字が **quantile**(T,s,e,r) の返す値となる．計算量は $\mathrm{O}(\log\sigma)$ 時間である．

Algorithm 9 **topk**(T, s, e, k): $T[s, e)$ 中で出現回数が多い文字順にその頻度とともに k 個返す

```
function topk(T, s, e, k)
    q は節点 t, 開始位置 st, 終了位置 en のタプル (t, st, en) を管理する優先度付きキューであり，優先度は en−st
    q.push(WT.Root, s, e)
    while isempty(q)= False do
        (t, st, en)←q.pop()
        if isleaf(t) then
            result.push(t, en−st)
            if result.size=k then
                break
            end if
        else
            zst=rank_0(B_t, st)
            zen=rank_0(B_t, en)
            ost=st−zst
            oen=en−zen
            if zen−zst>0 then
                q.push((child(t, 0), zst, zen))
            end if
            if oen−ost>0 then
                q.push((child(t, 1), ost, oen))
            end if
        end if
    end while
    return result
end function
```

- **topk**(T, s, e, k): $T[s, e)$ 中で出現回数が多い文字順にその頻度とともに k 個返す．

  Algorithm 9 に，擬似コードを示す．アルファベット木の中で優先度を頻度としたときの，優先度付き探索を行う．キュー Q を $Q:=\{\}$ と初期化

し，キューは節点を表すポインタ t と開始位置 st，終了位置 en のタプル (t, st, en) を管理する．はじめに根を示すポインタが r のとき，(r, s, e) を Q に追加する．この Q は優先度付きヒープであり，優先度は $en-st$，つまり対応する範囲が大きい順に取得できるとする．次に，Q から 1 つポップし，$B_t[s, e)$ 中に含まれる 0 と 1 の数を調べ，これらに対応する左の子と右の子をそれぞれキューに追加する．もしポップした要素が葉であった場合はそれを報告し，報告した個数が k に達したときに終了する．この操作の最悪計算量は $\mathrm{O}(\min(e-s, \sigma)\log\sigma)$ 時間である．しかし，要素間の頻度に大きな偏りがある場合は，無駄な節点を調べずに済み，計算量は $\mathrm{O}(k\log\sigma)$ 時間に近づく．

- **rangefreq**(T, s, e, x, y): $T[s, e)$ 中に出現する $x \le c < y$ を満たす文字 c の合計出現数を返す.
  根から x に対応する葉までのパスを p_x，同様に根から y に対応する葉までのパスを p_y とする．そして，p_x のパス中の節点で右側の子，および p_y のパス中の節点で左側の子はすべて報告対象のノードとなるので，これらのノードで該当する範囲内の文字の出現回数を報告する．p_x, p_y 中の節点の数，およびそれらの右側の子，左側の子の数の合計は高々 $4\lg\sigma$ 個であるので $\mathrm{O}(\log\sigma)$ 時間で求めることができる.
- **rangelist**(T, s, e, x, y): $T[s, e)$ 中に出現する $x \le c < y$ の各文字 c を頻度とともに列挙する.
  基本的な操作方法は先ほどの **rangefreq** とほとんど同じである．異なる点は p_x, p_y 中のパスから右側の子，左側の子をたどる際に，範囲が空にならない限り再帰的に節点を下の文字までたどって件数を列挙するという点である．計算量は報告される文字種類数を k としたとき，$\mathrm{O}(k\log\sigma)$ 時間である.
- **rangemaxk**(T, s, e, k): $T[s, e)$ 中に出現する文字を大きい順にその頻度とともに k 個返す.
  根から葉に向かって順にたどる際に，右側の子を優先してたどる．この際，対応する範囲が空集合になった場合は左側の子をたどるようにする．Algorihtm 9 の優先度付きヒープの優先度が $en-st$ の部分を，節点の辞

書式順位に変更したものに相当する．節点の辞書式順位とは，根からその節点までたどったときのビットを順につなげたビット列間の辞書式順位である．計算量は $O(k \log \sigma)$ 時間である．

- **rangemink**(T, s, e, k): $T[s, e)$ 中に出現する文字を小さい順にその頻度とともに k 個返す．
  先ほどの **rangemaxk** とは逆に左側の子を優先してたどる．
- **prevvalue**(T, s, e, x, y): $T[s, e)$ 中で $x \le c < y$ を満たす最大の c を返す．
  この操作は y の直前の値を求めることに相当する．**rangefreq** 操作と同様に，根から文字 x に対応する葉へのパス p_x と，根から文字 y に対応する葉へのパス p_y の間に存在する文字の中で最も p_y 側に存在する葉を探せばよい．パスで右の子をたどる際はそのまま y より小さいかという条件を満たしているかどうかをチェックし，満たしているのであればたどり，そうでないならば左の子をたどる．右の子をたどる際に，もし対応する範囲が空集合でなければそのままたどればよい．もし右の子をたどったときに対応する範囲が空集合，つまりその中には出現している文字がなくなったという場合には，代わりに左側の子の中で出現している最大の文字を探せばよい．計算量は $O(\log \sigma)$ 時間である．
- **nextvalue**(T, s, e, x, y): $T[s, e)$ 中で $x \le c < y$ を満たす最小の c を返す．
  先ほどの **prevvalue** と同様の操作で求められる．
- **intersect**(T, s_1, e_1, s_2, e_2): $T[s_1, e_1)$ と $T[s_2, e_2)$ の間で共通して出現する文字と頻度を返す．
  例えば，T が文書番号を格納しており，$T[s_1, e_1) = [7, 3, 2, 7, 5)$, $T[s_2, e_2) = [2, 1, 7, 2, 2, 7)$ がそれぞれ単語が出現している文書番号に対応した場合，**intersect**(T, s_1, e_1, s_2, e_2) は，これらの単語が共通して出現する文書番号を列挙して返す．文書番号，$T[s_1, e_1)$ 中の頻度，$T[s_2, e_2)$ 中の頻度のタプルで返すとする．両方で共通して出現している文書番号は 2 と 7 である．文書番号 2 は $T[s_1, e_1)$ では 1 回，$T[s_2, e_2)$ では 3 回出現しているので，$(2, 1, 3)$ を，同様に文書番号 7 については $T[s_1, e_1)$ では 2 回，$T[s_2, e_2)$ でも 2 回出現しているので，$(7, 2, 2)$ となり，結果として $\{(2, 1, 3), (7, 2, 2)\}$ を返す．アルゴリズムは次の通りである．はじめにキ

ューに (s_1, e_1, s_2, e_2) を登録する．次にキューから値を取り出し，それぞれの中での 0 の出現数と 1 の出現数を **rank** 操作によって求める．もし，0 側，1 側それぞれについて，$[s_1, e_1), [s_2, e_2)$ の中での出現数が 0 であればそれに対応する子の部分木の中での出現数は 0 であるのでその中には共通して出現する文字はないことがわかるので探索をそこで終了する．そうでなければ，子の対応する範囲をまたキューに入れて探索をする．

理想的には，共通して出現した文字に関係する節点のみが展開されるだけですむが，実際は内部節点のいくつかを無駄に展開してしまう．この場合の最悪計算量は，$O(e_1-s_1+e_2-s_1+1)$ であるが，例えば，alternation と呼ばれる複雑度を利用し，この **intersect** の回数を見積もることができることが知られている[10]．

5.4　さまざまなデータへの適用

前章で紹介したさまざまな操作は多くの問題へ適用が可能である．最も重要な文字列データへの適用は第 7 章や第 8 章の全文索引やテキストマイニングなどで深く扱っていくが，ここではその他の利用方法について紹介していく．

5.4.1　2 次元グリッド情報

2 次元中の異なる点集合 $V=\{v_i=(x_i, y_i)\}_{i=1}^n$，$x_i, y_i \in R$ が与えられた場合を考える．このとき，X 軸の値はすべて異なっていると仮定する．はじめに，これらの点集合を Rank 空間と呼ばれる $[0, n)\times[0, n)$ の整数に次のように変換する．$R_{V,x}(p)$ を V 中の p より x 軸の値が小さかった点の数，同様に $R_{V,y}(p)$ を V 中の p より y 軸の値が小さかった点の数とする．

$$R_{V,x}(a) = |\{x_i < a \mid (x_i, y_i) \in V\}| \tag{5.1}$$

$$R_{V,y}(b) = |\{y_i < b \mid (x_i, y_i) \in V\}| \tag{5.2}$$

このとき，点 (a, b) の Rank 空間中の値は $(R_{V,x}(a), R_{V,y}(b))$ として求められる．

このとき，V 中の点もそれぞれ Rank 空間中の値に置き換える，これによ

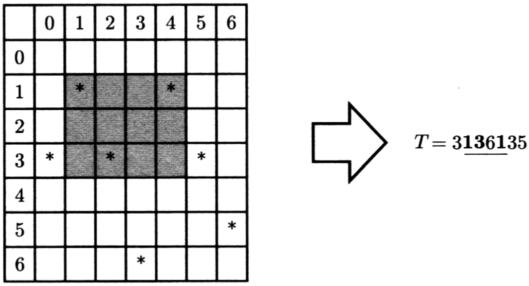


図 5.2 2 次元中の点集合に対する $1 \le x_i \le 4$, $1 \le y_i \le 3$ を満たす点の列挙

り，V の点集合は座標値が $[0, n) \times [0, n)$ の集合となり，またすべての X 軸の値はちょうど 1 回だけ出現するような点集合に変換される．

文字列 T を $T[R_{V,x}(x_i)] = R_{V,y}(y_i)$ のように定義すると，2 次元中の点集合は文字列 $T[0, n)$, $T[i] \in \{0, 1, \cdots, n-1\}$ で表すことができる．この Rank 空間への変換は V 中の点の X 軸と Y 軸の値をそれぞれソートしておき，それに対する二分探索を行うことにより，$O(\log n)$ 時間で求めることができる．

長方形領域 $[a, b) \times [c, d)$ の中に含まれる点の個数を列挙する問題を考えてみる．はじめに，この長方形領域を Rank 空間への変換を用いて $[x_{sp}, x_{ep}) \times [y_{sp}, y_{ep})$ の中に含まれる点をすべて列挙する問題へと変換できる．

この問題は先ほどの配列 $T[0, n)$ で考えれば，$T[x_{sp}, x_{ep})$ 中に含まれる y_{sp} 以上，y_{ep} 未満の値を持つ文字をすべて列挙する問題に還元できる(図 5.2)．出現数を求めるのであれば，**rangefreq**$(T, x_{sp}, x_{ep}, y_{sp}, y_{ep})$ を利用すれば，$O(\log m)$ 時間で求めることができ，出現している点を報告するのであれば，**rangelist**$(T, x_{sp}, x_{ep}, y_{sp}, y_{ep})$ によって，出現している全点を報告できる．

従来こうした 2 次元グリッド情報に対する長方形クエリを利用した計算量は最悪計算量が抑えられない，もしくは必要な作業領域量が抑えられないという問題があった．ウェーブレット木を利用した場合，値の種類数を m としたとき，計算量はどんな場合でも常に $O(\log m)$ 時間であり，また作業領域量はどんな場合でも $n \lg m$ ビットであり，もとのデータをそのまま表現した場合と変わらない．この他さらに長方形中の平均や，分散などを求めるといったこ

とも補助データ構造を利用すると可能となる[24].

5.4.2　グラフ情報

グラフ情報もウェーブレット木を利用して効率的に扱うことができる．ここでは有向グラフを考える．グラフ G の頂点数を n，枝数 m とする．はじめに，各頂点ごとに，それに隣接する頂点番号リストを格納し，それらを連結した配列を $S[0,m)$ とする．また，各頂点番号が S 中のどこから開始されたかを表すビット列 $B[0,m)$ を用意する．S を元にしてウェーブレット木を構築し，B には完備辞書を構築する．これらは $n\lg(m/n)+m\lg n+\mathrm{O}(n)+\mathrm{o}(m)$ ビットで格納できる[22].

このとき，頂点 v の i 番目に隣接している頂点は $l=\mathbf{select}_1(B,v)$ としたときに，$S[l+i-1]$ でアクセスできる．また，逆頂点 v に i 番目でつながっている頂点は $p=\mathbf{select}_v(S,i)$ としたとき，$\mathbf{rank}_1(B,p)$ で求めることができる.

5.5　ウェーブレット木の圧縮

ウェーブレット木はそのままでももとの文字列と同じサイズであるが，圧縮して扱うことができる．ウェーブレット木は 2 つの方法で圧縮できる．1 つめはハフマン木を利用する方法，もう 1 つはウェーブレット木の各ビット列を圧縮する方法である．これらについて順にみていく.

5.5.1　ハフマン木による圧縮

もとの文字列 T の各文字の頻度をもとにハフマン木を構築し，ハフマン木をアルファベット木として利用しウェーブレット木を構築する．このとき，各内部節点に付随するビット列の合計サイズは，ハフマン符号により出力される総符号長に一致するので，$nH_0(T)$ ビットである.

しかし，ハフマン木の高さは最悪 $\sigma-1$ となってしまう問題がある．これについては最大高さを $\mathrm{O}(\lg\sigma)$ で抑えつつ，全体のサイズを $n(H_0(T)+2)$ に抑えることができる方法がしられている[21]．また，もとの文字間の大小関係が失われてしまうので例えば **quantile** といった操作は実行できなくなる.

操作の際のクエリのアルファベットの出現分布が入力と同じ場合は，1文字あたりの平均計算量は $O(H_0(T))$ となる．

5.5.2　ビット列の圧縮

もう1つはアルファベット木の形はそのままで，各節点に付随するビット列を圧縮する方法である．

各内部節点 v に付随するビット列 B_v をそのままではなく，0次経験エントロピーのサイズで格納すれば，全体のサイズを $nH_0(T)$ に抑えることができる．

これを示すために，アルファベットが $\{a,b,c,d\}$ であり，根に付随するビット列中の0と1の数をそれぞれ n_0, n_1 とし，左，右の子に付随するビット列中の0と1の数をそれぞれ $n_{00}, n_{01}, n_{10}, n_{11}$ とした場合を考える．$n_{00}, n_{01}, n_{10}, n_{11}$ の数はそれぞれ，a,b,c,d の出現回数に一致する．

根 v に対応するビット列の0次経験エントロピー $nH_0(B)$ は，

$$nH_0(B) = n_0 \lg(n/n_0) + n_1 \lg(n/n_1) \tag{5.3}$$

である．次に，左の子，右の子に対応するビット列の0次経験エントロピー $nH_0(B_0), nH_0(B_1)$ はそれぞれ

$$nH_0(B_0) = n_{00} \lg(n_0/n_{00}) + n_{01} \lg(n_0/n_{01}) \tag{5.4}$$

$$nH_0(B_1) = n_{10} \lg(n_1/n_{10}) + n_{11} \lg(n_1/n_{11}) \tag{5.5}$$

である．

これらの合計サイズは，

$$\begin{aligned} &nH_0(B) + nH_0(B_0) + nH_0(B_1) \\ &\quad = n_{00} \lg(n/n_{00}) + n_{01} \lg(n/n_{01}) + n_{10} \lg(n/n_{10}) + n_{11} \lg(n/n_{11}) \\ &\quad = nH_0(T) \end{aligned} \tag{5.6}$$

である．ここでは $n_0 = n_{00} + n_{01}$, $n_1 = n_{10} + n_{11}$ を利用している．

これと同様に，一般の場合でも，各内部節点のビット列を0次経験エントロピーで格納できれば，全体サイズを，もとの文字列の0次経験エントロピー

でおさえられることが示せる．

5.6 アルファベット数が大きい場合

ウェーブレット木の問題点は，アルファベット数が σ のとき，保持するビット配列の数が $\sigma-1$ 個であることである．ビット配列の総長自体は $n \lg \sigma$ でおさえられるが，各ビット配列ごとにそれを格納するポインターを格納すると，$O(\sigma \log n)$ ビット必要であり，$\sigma \sim n$ の場合，ビット配列を保持するためのオーバーヘッドは $O(n \log n)$ ビットにもなる．このオーバーヘッドをなくす方法を紹介する．

5.6.1 ポインターがないウェーブレット木

最初に紹介する方法は，同じ高さの節点に付随するビット列をすべてつなげる方法である[3]．これにより，ビット列は $\sigma-1$ 個ではなく，$\lg \sigma$ 個にまとめられる．この場合，ある節点に対応するビット列は，その深さのビット列の一部に対応する．この対応する位置を保持せず，木を上下に移動する際に求めてしまうというのがアイディアである．

現在ある高さのビット列の $B[st, en)$ に対応している状態から 0 側の子に移動する場合を考えよう．この場合，ビット列の開始位置は変わらないが，終了位置は変わることになる．この終了位置は，$B[st, en]$ 中に存在する 0 の数と一致する．0 の個数は $n_z = \mathbf{rank}_0(en, B) - \mathbf{rank}_0(st, B)$ と求められるので，$en := st + n_z$ と更新すればよい．

次に 1 側の子に移動する場合を考える．この場合は st 側が更新される．これも同様に，n_z 個分だけ，もとの位置から移動するので $st := st + n_z$ と更新すればよい．このように，子に移動する場合は **rank** 操作だけで正しいビット配列の領域を求められる．

今度は，葉から親にたどっていくことを考える．それぞれの高さでの開始位置はそれぞれの文字の出現数を c_i としたとき，c_i を単進符号で記録しておけば **select** 操作で効率的に求めることができる．

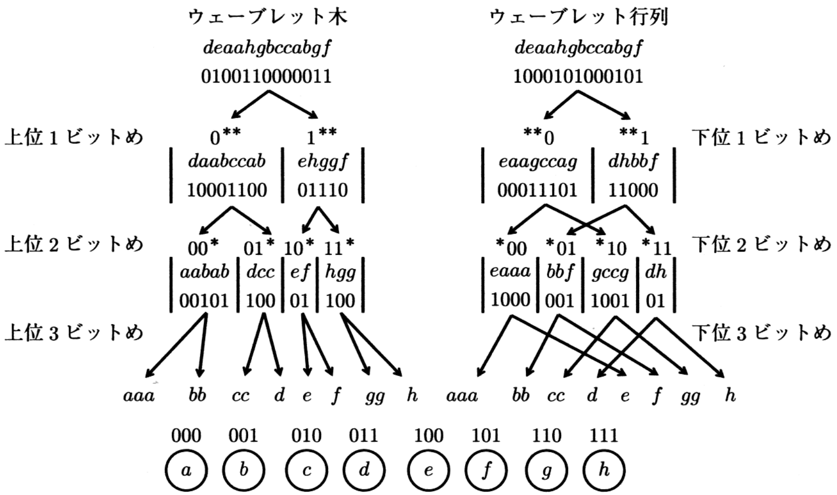


図 **5.3**　ウェーブレット行列の例

5.6.2　ウェーブレット行列

ウェーブレット木に代わるもっと自然な表現方法として，ウェーブレット行列[4]が提案された．ウェーブレット木では各節点の子孫は，ビット列で同じ領域に存在していた．例えば，深さ l の節点がビット列 $B_l[s,e)$ に対応している場合，この子孫のビット列においても，$B_k[s,e)$, $k>l$ の範囲に対応していることが求められた．しかし，この制約を外したとしても各節点の位置は次のように追跡できる．ウェーブレット行列では，同じ深さでビット列の 0 で遷移した節点の文字を並べた後に，1 で遷移した節点の文字を並べる．

これは，ウェーブレット木が最上位ビットから順にフィルタリングしているのに対し，ウェーブレット行列が最下位ビットから順にフィルタリングしているのに対応する．図 5.3 に例を挙げる．最初に，最下位ビットでフィルタリングを行う．これにより，a, c, e, g の文字が 0 側の子に，b, d, f, h の文字が 1 側の子にフィルタリングされる．次に 2 番めのビットで同様にフィルタリングを行う．これにより，a, b, e, f が 0 側の子に，c, d, g, h が 1 側の子にフィルタ

リングされる．最後は最上位ビットでのフィルタリングであるので，a, b, c, d が 0 に，e, f, g, h が 1 にフィルタリングされる．

ウェーブレット行列が優れているのは，5.6.1 節で扱ったポインターがないウェーブレット木とは異なり，途中で節点に対応する範囲を求める必要がないことである．$B_l[i]$ のビットの子をたどるときには，0 の場合は $\mathbf{rank}_0(B_l, i)$，1 の場合は $z_l + \mathbf{rank}_1(B_l, i)$ が対応する位置となる，ただし z_l は深さ l のビット列中に含まれる 0 の数である．

ウェーブレット行列での子への遷移は 1 回の **rank** 操作で実現できる．同様にして，親への遷移も 1 回の **select** 操作で実現できる．

また，ウェーブレット行列でも，各節点の位置が変わっただけで，節点が同じように存在していることに注意すると，ウェーブレット木で実現していた各操作はウェーブレット行列でもまったく同様に実現できる．

ウェーブレット行列は実装も容易であり，実用的にも優れている．実装の際は $\lg \sigma$ 個のビット列と，それぞれのビットでの 0 の数を格納するだけでよい．作業領域量は，$n \lg \sigma + 2(\lg \sigma)(\lg n)$ ビットである．

6

文字列データ圧縮

本章では，文字列に対するデータ圧縮法を紹介する．近年，大規模文字列データを扱ううえで，データ圧縮は特に重要な技術として注目されている．はじめに基本的なデータ圧縮を紹介したあと，LZ法を用いた圧縮，PPMを利用した圧縮，BWTによる圧縮をそれぞれ説明する．最後に，圧縮したまま任意の位置のデータにアクセスできる透過的データ圧縮を紹介する．

データ圧縮とは，データの内容や性質を保ったまま別の表現方法で表すことで，もとの表現より少ないデータ量で表現することである．

データ圧縮は古来より重要な技術であったが，近年大規模データを扱ううえで特に重要な技術として注目されている．なぜなら，データ圧縮を利用すれば，データを記録，通信するために必要なリソース量を減らせるだけでなく，データをより上位の高速な記憶装置で処理でき，全体の計算コストを小さくできるからである．

データ圧縮の実現と，データの理解は本質的に同じである．データを圧縮するには，データ中に存在する規則性や性質を理解し，よりデータに適した表現を選ばないといけないからである．しかし，多くの場合データがどのような規則性や性質を持つのかを前もって知ることはできない．与えられたデータと多少の事前知識のみからデータの規則性や性質をつかみとり，それに合わせてデータを表現するのがデータ圧縮の目標ともいえる．

圧縮の実用的な側面として，データをいかに小さく表現できるかだけでなく，いかに効率よく圧縮，復元できるか，またデータをすべて見てから圧縮するのではなく，前から順に圧縮できるかといったさまざまな目標が存在する．

従来は圧縮されたデータは一度復元しなければ扱うことができなかった．しかし，近年データを圧縮したまま任意の部分のデータのみを復元したり，ある文字列が出現しているかを調べたり，さまざまな統計量を求めるということが可能となっている．しかも，もとの文字列をそのまま扱うよりもさまざまな処理を効率的に実現できることが知られている．こうした面を踏まえれば，データ圧縮はより目的に沿ったデータ変換として捉えたほうが自然と考えられる．

6.1　基本的なデータ圧縮の例

第 2 章では符号を紹介し，各文字ごとの出現確率，またはその出現傾向がわかる場合に適切な符号を利用すればデータをもとより小さく表現できることを示した．

この章では文字ごとを独立に符号化するのではなく文字列の特徴や傾向を利用してどのように圧縮できるかについて見てみる．

6.1.1　連長圧縮

はじめに紹介するのは，単純だがよく利用される連長圧縮である．連長圧縮では，連続する文字を文字とその繰り返し回数のペアで記録する．この繰り返し回数のことを連長と呼ぶ．例えば，データ圧縮対象の文字列として $T=$ "*aaabbbbbaa*" を考えてみると，最初に "*a*" が 3 回，"*b*" が 5 回，また "*a*" が 2 回繰り返されることから S は "*a*3*b*5*a*2" と表現できる．復元の際は，前から順に文字と連長を調べて文字を復元すればよい．

この連長の符号化には第 2 章の正整数の符号を使うことができる．例えば，連長が小さいものが多いのであれば γ 符号や δ 符号など，それに合わせた符号を利用すればよい．連長圧縮は単純でありながら有効であり，多くのデータ圧縮の前処理，または後処理で利用されることが多い．

ここでは，任意の位置の文字に圧縮したままアクセス可能な連長圧縮を考えてみる．これは先ほどの連長の符号化に完備辞書を組み合わせればよい．文字列 T 中で先頭，もしくは前の文字と異なる $(T[i]\neq T[i-1])$ 場合は $B[i]=1$，そうではなく前の同じ文字である場合は $B[i]=0$ であるようなビット列 B を考える．そして B に対する完備辞書を構築する．また各繰り返しの先頭文字を U に格納する．例えば，$T=$ "*aaabbbbbaa*" の場合 $B=$ "1001000010" であり，$U=$ "*aba*" である．この際 $T[i]$ は $U[\mathbf{rank}_1(B,i+1)-1]$ として求められる．このような完備辞書を利用した連長圧縮のサイズは疎なビット列に対する完備辞書を利用した場合，T の長さが n，アルファベット種類数を σ，連長の数を r としたとき，$r\lg\sigma+r\lg(n/r)+2r$ ビットであり，定数時間でアクセスできる．

6.1.2　Re-Pair

次に Re-Pair と呼ばれる方法を初めに紹介する．文字列 T が与えられたとき，次の手順を繰り返す．

- T 中に含まれる文字の中で最頻である文字ペア $T[i]T[i+1]=ab$ を求める．
- 新しい整数番号を s とし，$s\rightarrow ab$ というルールを登録し，T 中の ab をすべて s に置き換える．

- T 中のすべての文字の出現頻度が 1 になるまでこの操作を繰り返す．

このようにして作られたルールは T のみを生成するようなルールであり，SLP (Straight-line program) と呼ばれる．ルールは配列や木などを利用して表現することができる．

6.2　辞書を用いた圧縮：LZ 法

次に辞書を利用したデータ圧縮について考えてみる．連長圧縮では直前の文字の繰り返しを効率的に符号化していたが，同じ部分文字列が繰り返される場合も多くみられる．例えば言語データでは単語やフレーズ，引用などは繰り返し出現する．辞書を利用した圧縮では繰り返し出現した文字列をそのまま出力する代わりに，以前に出現した文字列から構築された辞書中の番号を出力して圧縮を行う．辞書を用いたデータ圧縮は，同じ文字列が繰り返される場合に非常に高い圧縮効率を達成する．

例えばソースコードのバージョン管理の場合，各バージョンごとに前のバージョンの一部分だけが変更されたコードが格納される場合が多い．この場合，ソースコード全体をそのまま格納するのではなく，前のバージョンと違う箇所のみを記録すれば，少ないデータ量でバージョン管理をすることができる．また，同じ生物種間のゲノムは非常に似ており，ゲノム解析において大量の個体のゲノム情報を格納する場合に非常に効率的に格納できる可能性がある．この他，ログ情報や，規則性が強くみられる行動情報，コピーや引用が非常に多いウェブ上の情報なども同様に効率よくデータ圧縮できる．

辞書を用いたデータ圧縮は，復元が速いという特徴があるため，データの圧縮処理が一度だけで，圧縮したデータを配布するときなどに有利である．

本章では代表的な辞書型データ圧縮法である LZ 法を紹介する．

6.2.1　LZ 法

LZ 法は Lempel と Ziv らによって 1977 年と 1978 年に提案された．これら 2 つの手法はそれぞれ LZ77 法，LZ78 法と呼ばれている．

LZ 法では繰り返し出現した文字列を過去に出現した文字列のコピーとして

表現し圧縮を実現する．このとき，コピーを過去に出現した場所との相対距離，長さで表すのが LZ77 法であり，コピー候補となるフレーズを辞書に登録していき，コピーを辞書中の番号で表現するのが LZ78 法である．

また，LZ 法は増分分解法とも呼ばれる．なぜなら，LZ 法は入力テキストを逐次的にフレーズに分解しているとも考えられるからである．LZ 法の操作は単純ながら，文字列の経験エントロピーに漸近する符号長で記録でき，符号，復号ともに単純にできるため多くのデータ圧縮で用いられている．

6.2.2 LZ77 法

はじめに LZ77 法について説明する．LZ77 法では入力文字列 $T[0,n)$ をフレーズ列 $Z[0,n')$ に分解する．ただし $Z[i]$ は T 中の部分文字列に対応し，$T[0,n){=}Z[0]Z[1]\cdots Z[n'{-}1]$ と分解される．

この分解は次のように決定される．入力文字列の $k{-}1$ 文字めまで処理が進み $T[0,k{-}1]{=}Z[0,k'{-}1]$ と分解されているとする．このとき，次のフレーズ $Z[k']$ は，$T[0,k{-}1]$ 中から開始する部分文字列と最長一致する部分文字列と，その次の一致しない文字をつなげた文字から構成される．つまり $Z[k']{=}T[k,k{+}l{+}1]{=}T[p,p{+}l]c$, $0{\le}p{<}k$ であり，l は過去の最長一致数，c は不一致文字である．もし，文字 $T[k]$ が以前に出現していない場合は，過去に一致する部分文字列は見つからないので，一致長は 0 となり，フレーズは不一致文字だけからなる．

このとき各フレーズは (p,l,c) の 3 つ組で表される．p はそのフレーズが以前に出現した位置であり，l は一致長，そして c は不一致文字である．

図 6.1 に，$T{=}$“$abracadabra$” に対する LZ77 法による分解の例を挙げる．例えば，フレーズ番号が 5 であるフレーズは，$T[0,4)$ の “$abra$” が一致し，最後の文字 \$ が不一致文字である．

こうして作られたフレーズはすべて異なる．なぜなら，もし同じフレーズがあり，それらが順に $p_1\cdots p_2$ と出現しているならば，p_2 は p_1 を一致文字として次の不一致文字を含んだより長いフレーズを採用しているはずだからである．

このような文字列からフレーズへの分解に関して，次の定理が成り立つ

$T = abracadabra\$$

$[a]$ $[b]$ $[r]$ $[ac]$ $[ad]$ $[abra\$]$

0 1 2 3 4 5

番号	フレーズ	出現位置	一致長	不一致文字
0	a	*	*	a
1	b	*	*	b
2	r	*	*	r
3	ac	0	1	c
4	ad	5	1	d
5	$abra\$$	0	4	$\$$

図 6.1 LZ77 法の例

[17].

定理 6.1 アルファベット種類数が σ である入力文字列 $T[0,n)$ をフレーズ列 $T{=}Z[0,n')$ と分解し，$Z[0,n')$ の要素がすべて異なるとする．このとき任意の k について $n'\lg n'{=}nH_k(T){+}\mathrm{o}(n)$ ビットが成り立つ．また，この場合のフレーズ数 n' は，$\sqrt{n}{\leq}n'{\leq}n/\log_\sigma n$ を満たす． □

これは LZ77 法に限らず，すべてのフレーズへの分解に対して成り立つ．

この定理を利用し，LZ77 法によるデータ圧縮のサイズは次のように評価される．LZ77 法の各フレーズは位置 p と，長さ l と，不一致文字 c から構成される．位置を示す p の合計サイズは $n'\lg n{=}n'\lg n'{+}n'\lg(n/n'){=}nH_k(T){+}\mathrm{o}(n)$ ビットである．また，長さ情報 l については，長さ n，重み n' の完備辞書で表現すれば，$n'\log(n/n'){+}2n'$ ビットで格納でき，不一致文字 c は $n'\lg\sigma$ ビットである．これらを合計すると $nH_k(n){+}\mathrm{o}(n\log\sigma)$ ビットで格納することができる．

LZ77 法を実行する際に問題となるのは，過去の文字列から最長の一致文字列を探してくる部分である．これまでに，接尾辞配列などを用いることで $\mathrm{O}(n)$ 時間で LZ77 分解を求める方法が知られている．しかし，この方法では文字列全体が前もって与えられた後にしか，LZ77 分解を求めることができな

い．一方，データ圧縮のアプリケーションによっては，入力文字列を順に調べ最長の文字列を調べる必要がある．よく利用される方法は滑走窓を利用した方法である．これは，適当な窓幅 w を決め，i 文字めを処理しているときは $T[i-w,i)$ の履歴の中から一致文字列を探す方法である．この滑走窓は巡回して保持されており，更新時間は定数時間である．これに加えて，先頭数文字を利用して求められたハッシュ値で一致文字列が出現している候補位置を調べ，そこから実際に何文字一致しているのかを調べる．この方法では最長一致文字列が見つかる保証はないが，実用的には十分長い一致文字列を見つけることができる．

LZ77 法ではすべてのフレーズが異なるように，最後に不一致文字をフレーズに含めるようにしていた．しかし，このようなフレーズ分解は同じ文字列が繰り返される場合に，必ずしも効率的な表現であるとは限らない．実用上の圧縮率をより高めた LZSS と呼ばれる方法では滑走窓を利用し，不一致文字はフレーズに含めず，さらに一致文字列が見つかったかどうかを 1 ビットで記録する方法を採用する．このとき不一致文字は出力しない．コピーの方がもともとの表現よりオーバーヘッドが大きくなる場合にはコピーを行わずに，そのままの文字を出力する．LZSS は多くのデータ圧縮ソフトで使われている．

6.2.3　LZ78 法

LZ78 法でも LZ77 法と同様に入力テキスト T をフレーズに分解するが，フレーズの集合から構成される木を逐次的に構築する点が異なる．

入力文字列 $T[0,k)$ まで処理し，$Z[0,k')$ が得られていたとする．このとき，フレーズ集合 Z から構成される木を保持する．この木では各枝に文字が付随する．そして，根から葉を含む各節点までのパスが各フレーズに対応する．このような木は Trie と呼ばれ 7.6.1 節で詳しく扱う．次に k の位置からはじまる部分文字列の中で，これまでのフレーズの中で最長一致するフレーズを求め，そのフレーズ番号を出力する．これは先ほどの木の中の節点番号でよい．そして，次の不一致文字を記録し，それを木に追加する．この操作をすべての文字について順に適用する．1 つのフレーズを処理するたびに木のサイズは 1 つずつ増えていく．最終的に，フレーズの集合から構成される木と，フレーズ

$T = abracadabra$
$[a]$ $[b]$ $[r]$ $[ac]$ $[ad]$ $[ab]$ $[ra]$

番号	フレーズ	一致フレーズ番号	不一致文字
0	a	*	a
1	b	*	b
2	r	*	r
3	ac	0	c
4	ad	0	d
5	ab	0	b
6	ra	2	a

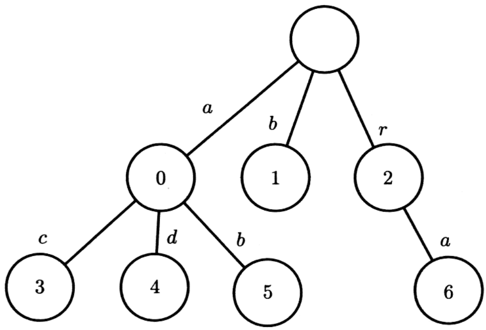


図 6.2　LZ78 法の例

番号の列が得られる．

図 6.2 に T=“$abracadabra$” に対する LZ78 法による符号化の例を示す．はじめに空の木に対して空文字しかマッチしないので “a” がフレーズとして抽出され，木に “a” だけからなる文字を追加する．文字 “b” と “r” についても同様にその文字だけからなるフレーズを追加する．その次に “ac” を追加するときは，“a” は一致するのでフレーズ番号 0 を出力し，“c” を追加する．

各節点には，それに対応する文字とフレーズ番号が記録される．LZ77 法の場合は最長一致文字列を探すためには別のデータ構造が必要となるが，LZ78

法は，木を利用してどのフレーズを使えばよいかを調べられる．

LZ78 法の場合も，LZ77 法のときと同様にすべてのフレーズが異なる保証があるので，フレーズ番号を格納したサイズは $nH_k(T)+o(n)$ ビットであり，不一致文字のサイズは $o(n\log\sigma)$ ビットであるので全体で，$nH_k(T)+o(n\log\sigma)$ ビットである．

LZ77 法と LZ78 法を比較した場合，長い繰り返しなどが見られる場合 LZ77 法の方が高い圧縮率を達成することが知られている．しかし LZ77 法の場合は，圧縮の際に最長一致部分文字列を探索する部分で工夫が必要である．

6.3　文脈を利用した圧縮：PPM 法

文脈から，次の文字を予測して符号化することで圧縮する PPM 法を紹介する．PPM(Prediction by Partial Matching)法では，どの文字が出現しそうかを過去の文脈を利用して予想し，各文字の出現確率を推定する．そして，その確率に基づいて符号を構成し次の文字を符号化する．この圧縮は賭けと同じであり，もし予測がうまくいけば短い符号長を割り当てることができるが，予測が外れれば長い符号長が割り振られ，場合によってはもとの入力よりも長い符号長が割り当てられる．

復元の際は，圧縮時と同じように予測を行い，各文字の符号を構成し，読み込んだ符号に基づいて文字を復元し予測モデルを構成する．

辞書を利用したデータ圧縮では圧縮に比べて復元は高速に行えたが，PPM 法の場合は復元時にも圧縮と同じ予測を行うため，復元も遅い．しかし予測モデルを精緻に作ることで，高い圧縮率を達成することが可能である．

PPM 法では，いかに次の文字の出現確率を正しく予測できるかどうかが圧縮が，うまくいくかの鍵となる．文字列 $T[0,n)=c_0c_1c_2\cdots c_{n-1}$ の生成確率 $P(T)$ は，チェインルールによって次のように分解できる．

$$\begin{aligned}P(T) &= P(c_0, c_1, c_2, \cdots, c_{n-1}) \\ &= P(c_0)P(c_1, c_2, \cdots, c_{n-1}|c_0) \\ &= P(c_0)P(c_1|c_0)P(c_2, \cdots, c_{n-1}|c_0, c_1)\end{aligned}$$

$$= \prod_{i=0}^{n-1} p(c_i|c_0c_1\cdots c_{i-1}) \tag{6.1}$$

つまり，各文字ごとにそれまでに出現した文字列による条件付確率 $P(c_i|c_0c_1\cdots c_{i-1})$ の積に分解できる．

しかし，そのまま実際に出現した頻度に基づいた経験確率を利用して出現確率を推定することはできない．なぜなら，i 文字めを予測するときに利用する文脈は過去に一度も出現していないからである．なので，PPM 法では実際には直前の数文字だけを文脈として利用する．長い文脈であればあるほど，予測性能は高くなると考えられるので，PPM 法ではできるだけ長く一致した過去の文字列に基づいて確率を求める．

例に基づいて PPM 法を説明する．現在 T の i 文字め $T[i]$ を処理するとしよう．このときの文脈は $C{=}T[0,i)$ である．C の中で $T[i{-}d,i)$ と可能な限り長く一致する文字列を探す．次にその直後に各アルファベットが何回出現していたかどうかを調べる．その結果各文字 c の頻度が $f(c)$ であり，合計は n だとする．この場合，次の文字は $f(c)/n$ の確率で出現しそうだと考えられる．例えば，“*abcxbczabcybcxabcxyabc*” の次の文字を予測する場合，過去に最も長く一致する文字列は “*abc*” であり，その後に出現した文字は “*x*” が 2 回，“*y*” が 1 回であるので，“*x*” の出確確率を 2/3, “*y*” の出現確率を 1/3 とする．確率が決まれば，例えば算術圧縮などを用いて符号化することができる．

この方法の問題は過去の文脈では出現しなかった新しい文字が出現した場合に確率は 0 と設定されてしまい，符号化，復号化ができないことである．例えば先ほどの例で，“*z*” が出現した場合，“*z*” の出現確率は 0 であるため符号化できない．

そこで．次の文字が予測確率表には出現しなかったことを表すエスケープ文字 “*esc*” を用意し，エスケープ文字に対しても確率を割り当てておく．その分，他の文字の出現確率は低くする．例えば，エスケープ文字は常に 1 回分と考える場合，“*x*”, “*y*”, “*esc*” の出現確率はそれぞれ 2/4, 1/4, 1/4 となる．そして，“*x*”, “*y*” 以外の文字が出現した場合はエスケープ文字を符号化する．次に他の文字の出現確率を求めるために，予測に用いる文脈長を短くする．文

脈長を短くすると，文脈の出現回数も増え，より多くの種類の文字が出現しているのでそれらを用いて予測することができる．ここでも見つからなければさらに一致文脈を短くする．この操作は文脈長が 0 になるまで繰り返し，それでも見つからない場合はすべての文字が一様に出現する確率分布を利用して予測を行う．これにより，どのような文字が出現した場合でも正しく符号化できる．復号の際も符号化の際と同じように行い，復号された文字がエスケープ文字ならば文脈を短くして確率分布を再設定する．

先ほどの例の場合，最も長い一致文脈は "abc" であったが見つからない場合は 1 文字短い，"bc" を文脈として用いる．すると，"x" が 3 回，"y" が 1 回，そして "z" が 1 回出現しているので，それにあわせて出現確率を設定する．

まとめると，$P(c_i|c_0c_1\cdots c_{i-1})$ の確率は次のように決められる．

$$P(c_i|c_0c_1\cdots c_{i-1}) = p(c_i|c_{i-d}c_{i-d+1}\cdots c_{i-1})\prod_{j=1}^{d} P(esc|c_{i-d+j}\cdots c_{i-1}) \quad (6.2)$$

となる．ただし d は初めに c_i が見つかったときの分脈長である．

この際問題になるのは，エスケープ文字の出現確率をどのように設定するかである．これに対しては次の表に挙げた方法が提案されている．ただし，各記号の意味は次の通りである．

手法名	$P(esc\|C)$	$P(c\|C)$
A	$\frac{1}{n+1}$	$\frac{f(c)}{n+1}$
B	$\frac{\sigma}{n}$	$\frac{f(c)-1}{n}$
C	$\frac{\sigma}{n}$	$\frac{n-\sigma}{n}\frac{f(c)}{n}$
D	$\frac{\sigma}{2n}$	$\frac{2f(c)-1}{2n}$
X	$\frac{\sigma_1}{n}$	$\frac{n-\sigma_1}{n}\frac{f(c)}{n}$
C′	$\frac{\sigma}{n+\sigma}$	$\frac{f(c)}{n+\sigma}$

- n: 文脈 C の直後に出現した文字総数
- σ: 文脈 C の直後に出現した文字種類数
- σ_1: 文脈 C の直後に 1 回だけ出現した文字種類総数
- $f(c)$: 文脈 C の直後に文字 c が出現した回数

どの方法がよいかはデータの種類に依存するが，一般的に Method A は予測精度が低く圧縮率がよくないのに対し，Method D や Method X が予測精度が高く圧縮率が高いことが報告されている．

PPM 法については，より高い精度で文字を予測するような方法も多く提案されている[30]．

言語モデルとの関係

PPM 法と同じような問題は自然言語処理における言語モデルで見られる．言語モデルの場合は，得られた文字列に対して確率を与える問題であり，文法的，用法的に正しい文に対しては高い確率，そうでない文に対しては低い確率を与えるのが目的であり，機械翻訳や音声認識などで利用されている．言語モデルとの違いは，言語モデルはコーパスなどを用いて確率モデルを前もって構築しておくのが普通なのに対し，PPM の場合は符号化，復号化の際に動的にモデルを作成する点である．しかし，近年自然言語処理でも動的にモデルを構築する方法も多くなるほか，PPM の場合でもあらかじめ作っておいたモデルを併用するなど，両者の考えは急速に近づいている．今後，両方の分野で生まれた技術がお互いに使われることが期待される．

6.4 BWT を利用した圧縮

第 3 章で扱ったように，BWT は文字列を圧縮しやすい文字列に可逆変換を行うが，変換後の文字列をどのように圧縮するかについては，さまざまな研究が提案されている．

BWT が提案された論文[2]で説明された圧縮手法について紹介する．アルファベット集合 $\Sigma=\{0,1,2,\cdots,\sigma-1\}$ からなる長さ n の文字列 T とし，BWT 後の文字列を T_B とする．この後，T_B に対し，次の 3 つの変換を適用する．

- 先頭移動法により $T_M{=}mtf(T_B)$ に変換
- 連長圧縮により $T_R{=}rle(T_M)$ に変換
- T_R をエントロピー符号化

それぞれの操作について以下で解説する．

先頭移動法

はじめに配列 $U[0,\sigma)$ を用意する．表は $U[i]{=}i$ と初期化する．先頭移動法では，配列 U に対し $p{=}\mathbf{access}(U,c)$ は c が U 中の何番目の文字であるかを返すと同時に，参照した文字を U の先頭に移動し，それ以外の文字を 1 つずつ後ろにずらす操作だとする．入力文字列の各文字に対して **access** 操作を順に適用し，$T_M[i]{=}\mathbf{access}(U,T_B[i])$ とする．T_M から T_B を復元する場合は，$T_B[i]{=}U[T_M[i]]$ とし，$\mathbf{access}(U,T_m[i])$ を行うことで入力のときと同じように先頭移動法の表が更新される．

同じ文字が近い位置で繰り返し出現しやすいような文字列に対し先頭移動法を適用すれば，文字列を小さい値が頻出するような文字列に変換できる．例えば，$U{=}[a,b,c]$ であり，$T{=}$“$aaabbbbcaaaa$” の場合，先頭移動法による変換後の文字列は “000100022000” となる．先頭移動法により，任意の文字が連続しやすい文字列から，小さい文字が出現しやすい文字列に変換される．

連長圧縮

次に，変換された文字列に対して，連長圧縮を適用する．特に多くの 0 が連続して出現することが期待されるので，効率よく圧縮できる．

エントロピー符号化

最後にエントロピー符号化を適用する．全体の文字の頻度表を用いて，ハフマン符号，もしくは算術符号などを用いて符号化する．このようにして BWT 後のテキストを符号化できる．LZ 法とは違って，BWT 後のテキストでは局所的な冗長性に注目して圧縮すれば，十分な圧縮率を達成できるという点に注意する．

この手順による BWT の圧縮は実用的にも高い圧縮率を誇るほかに，理論

的にも k 次経験エントロピーに漸近することが示されている[14].

6.5　透過的データ圧縮

データ圧縮の中で近年急速に進歩したのが透過的データ圧縮である．従来のデータ圧縮では，データを圧縮した後は一部分しか使わない場合でも，全体を一度復元してからのみ利用できた．しかし，近年大規模データを扱うようになるにつれ，圧縮したまま全体を復元しなくてもデータを参照可能な手法が登場してきた．こうしたデータ圧縮は，ユーザーから見れば透過的にデータをアクセスできるため，透過的データ圧縮と呼ばれる．

6.5.1　透過的符号

はじめに透過的にアクセスできる符号を紹介する．これまで紹介してきた符号は基本的には瞬時復元可能であり，前から読んでいったときに瞬時に復元できる符号であった．それでは，全体を復元せず任意の位置の符号を復元できるような符号は考えられるだろうか．

可変長符号の問題点は途中の符号を復元しようとしたとき，その符号がどの位置から始まるかわからないことである．しかし，すべての開始位置をそのまま記録すると符号と同じか，それ以上のサイズが必要となってしまう．

そこで k 個の符号ごとにブロックに分割し，各ブロックごとに，符号の開始位置を二進表現でそのまま記録する．ブロックの数は n/k 個なので，全体で $(n/k)\lg n$ ビット必要となる．復号時は，対応するブロックのみを復元し，該当する値を取り出す．この場合，計算量は $\mathrm{O}(k)$ である．例えば，$k=\log^2 n$ とすれば，符号の他に必要な作業領域量は，$n/\lg n=\mathrm{o}(n)$ ビットであり，アクセスには $\mathrm{O}(\log^2 n)$ 時間かかる．また，開始位置を完備辞書で記録し参照することもできる．しかし，密なビット列に対する **select** 操作は **rank** や **access** 操作と比べて遅いという問題がある．

実用的に高速に処理できる透過的な符号を紹介する[1]．これから符号化する正整数列の二進表現を並べたものを $v_1v_2\cdots$ とする．また，それぞれの二進表現の長さを $l_i=|v_i|$ とする．適当な正整数 $k_0>0$ を用意し，$l_i<k_0$ であ

れば $B_0[i]=1$，そうでなければ $B_0[i]=0$ となるようなビット配列 B_0 を用意する．つまり，B_0 は l_i が k_0 ビットのみで表現できるかどうかを表す．そして，$A_0[i]$ に v_i の下位 k_0 ビットを保存したような配列 A_0 を作る．そして，k_0 ビットで表現できなかった符号の残りのビット列から符号列を構成し，それらに対し再度，k_1, B_1, A_1 を使って符号化をする．この操作をすべての値が表現できるまで繰り返す．

このようにして，もとの符号列 $v_0v_1v_2\cdots$ は，ビット配列の集合 B_0, $B_1, \cdots, B_r$ と，配列の集合 $A_0, A_1, \cdots, A_r$ と，ビット幅の集合 $k_0, k_1, \cdots, k_r$ で表現される．

これらの配列を用いてどのように復元できるかを見てみよう．例として，p 番目の符号 v_p を復元する場合を考えてみよう．はじめに $B_0[p]$ を調べ，もし $B_0[p]=0$ ならば $A_0[p]$ を出力して終了する．そうではなく $B_0[p]=1$ ならば，$A_0[p]$ を下位 k_0 ビットとし，$p:=\mathbf{rank}_1(B_0, p)$ としたうえで，B_1, A_1 に対して今の操作を繰り返していく．

以上の操作では，**rank** 操作しか行わず，また A_i は固定長の配列なので定数時間でアクセスでき，$O(r)$ 時間で各値にアクセスできる．

6.5.2 統計的なブロック符号化

圧縮後のサイズを $nH_k+o(n)$ ビットとしたうえで，連続する $O(\log n)$ ビットの復元を定数時間で実現できる方法[11]を紹介する．

各文字を直前の k 文字で条件づけられた確率で符号化する．このときのサイズは $nH_k(T)$ ビットである．先頭に k 文字を追加するので，そのオーバーヘッドは $k\lg\sigma$ ビットである．それにより全体サイズは $nH_k(T)+2+O(k\log n)$ ビットを達成できる．対象文字列をブロックに分割し，各ブロックを独立に圧縮する．各ブロック B_i の先頭 k 文字はそのままの文字を出力し，残りの各文字を直前の k 文字で条件付けして符号化する．ブロック長を $b=\log_\sigma n$ としたとき，各ブロックの先頭部分をそのまま出力する部分を合わせたサイズは $O((n/b)k\log\sigma)$ ビットである．各ブロックは先頭の k 文字と，残りを算術符号で符号化したビット列のペアからなる．もし，圧縮した結果が $\lfloor 1/2\log n\rfloor$ ビット以上となる場合，圧縮せずにそのまま出力するようにする．

圧縮したビット列からもとの文字列への対応表の大きさは $o(n)$ ビットに抑えることができ，全体のサイズを $nH_k(T)+o(n)$ ビットで表現できる．復元する場合は対応するブロックを表引きで復元すればよいので，定数時間で復元できる．

6.5.3　ブロックベースの圧縮

次に紹介する方法では二進符号と表引きにより透過的データ圧縮を実現する[8].

アルファベットサイズが σ である文字列 $T[0,n)$ を考える．はじめに T を長さ $b=(\lg_\sigma n)/2$ のブロック $\{B_i\}_{i=1,\cdots,n/b}$ に分割し，これらの中で異なるブロックからなる集合を S とする．S のサイズは $O(\sigma^b)=O(n^{1/2})$ である．

次に，S を頻度の大きい順にソートする．ブロック S_i の順位を $r(S_i)$ とする．また，$r^{-1}(j)$ をこの逆関数とし，順位が j 番目のブロックの番号を返すとする．

各ブロック S_i の符号 $enc(i)$ を $r(S_i)$ の二進符号であるとする．これらの符号を順につなげたものを $V=enc(1)enc(2)\cdots enc(n/b)$ とする．また r^{-1} を $O(\sigma^b)$ 個のエントリからなる表に格納する．必要な作業領域は $O(\sigma^b \log n)=o(n)$ ビットである．また V のサイズは nH_k でおさえることができる．各 $enc(i)$ は固定長符号ではなく，接頭辞符号でもないため V のみからはもとの $r(S_i)$ を復元できない．

最後に，圧縮された各ブロックとその復元を定数時間で実現するために，索引を作成する．ブロックを $\log n$ 個ごとまとめ，これらをスーパーブロックと呼ぶ．そして各スーパーブロックの開始位置，スーパーブロック内の各ブロックの開始相対位置を記録する．これらの情報は $o(n)$ ビットで格納でき，各ブロックの先頭位置，およびそのブロック長は定数時間で求めることができる．

6.5.4　LZEnd 法

LZEnd 法[18]は，LZ77 法の変種と簡潔辞書を組み合わせることにより透過的アクセスを可能としつつ，優れた圧縮効率を達成できる．

LZEnd 法では LZ77 法の分解を次のように変更する．LZ77 法のときと同

様に $T[0,i)$ まで処理され，それらがフレーズ $Z[0,p)$ のように分解されていると仮定しよう．このとき LZ77 法では過去の文脈すべてから一致する部分文字列を探していた．LZEnd 法では一致した文字列の最後がフレーズの最後と一致するという制約を加える．つまり，$T[i,i')$ が，$Z[0]\cdots Z[q]$, $(q<p)$ の接尾辞と一致するような最長の部分文字列を探し，それと次の文字を加えたのをフレーズとする．

LZEnd 法ではフレーズは過去のフレーズ番号 q，一致長 l，そして不一致文字 c の 3 項 (q,l,c) から構成される．これら 3 項をそれぞれ別々にまとめた配列を次のように用意する．

- $source[0,n')$: フレーズ番号 q を並べた配列
- $B[0,n)$: フレーズが終了した位置を記録した完備辞書
- $char[0,n')$: 不一致文字を並べた配列

これら 3 つの配列により，例えば p 番目のフレーズの 3 項は

$$(source[p], \mathbf{select}(B,p+1)-\mathbf{select}(B,p)-1, char[p]) \tag{6.3}$$

として表される．LZEnd 法による分解はすべてのフレーズが異なることを保証できるため $source$ は $n'\lg n'=nH_k(T)+\mathrm{o}(n)$ ビットで記録できる．また，$char$ は $n'\lg\sigma$ ビット，B は $n'\lg(n/n')+2n'$ ビットで格納できるので合計で $nH_k(T)+\mathrm{o}(n)$ ビットで格納できる．

LZEnd 法では T 中の任意の連続する l 文字の復元を $\mathrm{O}(l)$ 時間で実現する．各フレーズの末尾の 1 文字は $char$ として直接格納されている．復元処理時には以前の参照元となったフレーズをたどり，そのたびに末尾の 1 文字を復元でき，たどる回数は l 回であるからである．

LZEnd 法の圧縮率は LZ77 法に匹敵しつつ，復元時の計算量は常に一定におさえられるという優れた性質を持つ．

7

全文検索

全文検索とは，対象とする文書の中から特定の文字列を検索することであり，文字列解析の中心の技術である．全文検索においては，大規模の文字列から特定の文字列を高速に，漏れなく，少ないリソース量で検索することが求められる．本章では，全文検索の技術として逐次検索，索引を利用した全文検索，そして索引の中でも圧縮技術と融合した圧縮全文索引を紹介する．

全文検索とは，検索対象の文書とクエリ文字列が与えられたとき，クエリが文書中に何回出現しているか，そして，どこに出現しているかを求める問題である．

例えば，検索対象の文書が $T=$“$abracadabra$”，クエリ文字列が $Q=$“a”，“$abra$” のとき，それぞれの出現位置は $\{0,3,5,7,10\}, \{0,7\}$ である．

この全文検索に対するアプローチは大きく 2 つに分けられる．1 つめはクエリが与えられるたびに，どこで出現しているかを文書すべてに対して調べる方法であり，逐次検索と呼ばれる．もう 1 つは，与えられた文書を分析して前もって索引を構築しておき，検索時はこの索引を利用して調べる方法であり，この索引を全文索引と呼ぶ．

前者の逐次検索は，毎回文書全体を調べる必要があるので，検索時の計算量が文書サイズに比例するという問題がある．しかし，検索対象文書が追加されたり，変更されたとしても索引を作り直す必要がないという利点を持つ．そのため，検索対象の文書が高頻度で変更される場合に，逐次検索は有効な方法である．それに対し，後者の索引を使う場合は，毎回文書全体を調べる必要はなく，大規模な文書に対しても高速な検索を実現できる．一方，文書の追加や変更，削除が発生するたびに，索引もそれに合わせて更新する必要があり，高頻度に更新される文書に対応するには工夫が必要になる．

また，文書に加えて索引も保存しなければならない．全文索引はもとの文字列と同じ情報を持っている．例えば，すべての 1 文字からなるクエリ文字列を検索して出現位置を求め，それらをつなぎあわせればもとの文字列を索引のみから復元できる．全文索引の容量は文書自体の容量と同等か，それより大きくなってしまう場合もある．

そのため，圧縮全文索引と呼ばれる新しい索引が登場している．圧縮全文索引は，全文索引でありながら，索引容量がもとのテキストより小さくできる特徴があり，さらに多くが検索時にもとのテキストを必要としない Self-Indexing と呼ばれる性質を持つ．つまり，圧縮全文索引はデータ圧縮と全文索引の両方の性質を持っているといえる．

本章では逐次検索について簡単に紹介した後に，全文索引を利用する方法，そして圧縮全文索引を利用する方法を紹介する．

7.1 問題設定

検索対象のテキストを T とし，$n{=}|T|$ とする．また，検索したいクエリ文字列を Q とし，$m{=}|Q|$ とする．T, Q の文字列はどちらもアルファベット集合 Σ の文字から構成されているとし，$|\Sigma|{=}\sigma$ とする．

このとき，次の問題を考える．

定義 7.1（**occ, locate**） 検索対象文字列 T とクエリ文字列 Q が与えられたとき，**occ, locate** を次のように定義する．

- **occ**(T, Q): T 中の Q の出現回数を返す．
- **locate**(T, Q): T 中の Q の出現位置の集合 $O{=}\{p_i\}$ を返す．

□

ただし，出現位置は開始位置であるとし，T 中の Q の出現位置が o ならば，$T[o, o{+}m){=}Q$ であることを意味する．

出現している位置がすべてわかれば，出現回数は瞬時にわかるので，**locate** を求める問題は **occ** を求める問題より難しい問題である．

また，本書ではクエリの出現箇所が重なっているときも，それぞれ別々の出現として扱うこととする．例えば $T{=}$“$aaabaaa$” で，$Q{=}$“aa” の場合，**occ**$(T, Q){=}4$, **locate**$(T, Q){=}\{0, 1, 4, 5\}$ である．

7.2 逐次検索

はじめに，索引を利用しないで検索する逐次検索について説明する．この場合検索対象テキストではなく，クエリ文字列を分析していかに効率的に探索を行えるかが問題となる．以降では主に **locate**(T, q) を求める方法を紹介する．

最も単純な方法は，T の各位置について，クエリ Q が実際に出現しているかどうかを 1 つずつ文字列比較を行い調べる方法である．この場合，T の各位置で，Q とあっているかどうかの比較を行うので，全体で高々 nm 回の文字比較が必要となる．それにより，**occ**(T, Q)，**locate**(T, Q) はともに $O(nm)$ 時間で求められる．これに対し，クエリ文字列 Q のみを分析して，あらかじ

め表を構築しておくことで高速化を図る．

はじめにBM(Boyer-Moore)法を紹介する．BM法ではクエリ文字列を検索対象文字列の各位置であっているかどうかを順に調べる．ここで$T[i,i{+}m)$と$Q[0,m)$が一致するかを調べて，$T[i{+}e]{\neq}Q[e]$であり，一致してなかった場合を考える．このときの$c{=}T[i{+}e]$を失敗文字と呼ぶ．この情報を利用し，$Q[0,m)$中で文字cが含まれているかどうかを調べる．例えばまったく含まれていないのであれば$T[i{+}e]$を含む場所ではクエリ文字列は一致しないので，$i{+}e{+}1$文字め以降から比較を行えばよいことがわかる．

この失敗文字cがクエリで最初に出現している位置までは，TとQは一致しない．よって，その位置までT側の位置をずらすことができる．この最初に出現する位置はあらかじめクエリを調べ，アルファベットごとに最初に見つかる位置を事前に計算して表を作っておけばよい．

BM法は最悪比較回数はnm回のままであるが，最適な場合の比較回数はn/m回まで減らすことができる．

また，QuickSearch法と呼ばれる方法では，$T[i,i{+}m)$との比較が終わり文字列比較位置をずらす際，比較対象の文字列の次の文字$T[i{+}m]$を調べ，それがクエリに含まれている位置まで移動する．$T[i{+}m]$の文字がクエリに含まれない場合，$m{+}1$文字スキップできるので，最適な場合の比較回数は$n/(m{+}1)$となり，BM法よりも少ない比較回数で文字列比較を行うことができる．その他の逐次検索の手法については[19]を参照してほしい．

7.3　全文索引

次に索引を利用した全文検索について説明する．全文索引を利用した検索では，あらかじめ検索対象の文字列を分析して索引を構築しておき，検索時にはこの索引を利用して検索を行うことで高速化を図る．

全文索引は次のような指標で性能を図ることができる．

- 検索時の計算量
- 索引の構築に必要な計算量と作業領域量
- 索引のサイズ

• 更新性能

一般に，検索時の計算量とその他はトレードオフの関係にある．検索対象テキストの情報を多く利用した索引は検索時の計算量を低くできるが，索引の構築コスト，サイズ，更新コストは大きくなる．一番極端な例が逐次検索であり，検索時の計算量は大きいが，索引に関するコストは存在しない．もう一方の極端な例が接尾辞配列や FM-index であり，これらは検索時の計算量は非常に小さく検索対象のテキストサイズに依存しないが，索引の構築，保持，更新コストは大きくなる．扱う問題に応じて適切な索引方式を選ぶ必要がある．

7.3.1 転置索引

はじめに，最も多く利用されている全文索引である転置索引について説明する．

転置索引(Inverted file index, Posting List とも呼ばれる)は，タームと呼ばれる部分文字列ごとに，それらがどの位置に出現したかを記録した出現位置リストを保持する(図 7.1)．タームとしては一般的には単語が選ばれるが，複数の単語からなるフレーズが選ばれたり，あるルールにマッチするような部分文字列を利用する場合もある．また，文書集合を扱う場合には，どの文書番号に出現したかのみを記録する場合と，文書番号とその中での出現位置の両方を記録する場合がある．

この転置索引の考え方は，書籍の末尾のキーワード集におけるキーワードとその出現ページと同じ考え方である．各文書ごとにどのタームが出現しているかを記録するのではなく，各タームごとにどの文書で出現しているのかを記録する．文書からタームではなく，タームから文書をひくため転置索引と呼ばれる．

転置索引では入力クエリが，タームに一致した場合は，**occ** はタームに紐付いている出現位置リストの長さとして求められ，**locate** は出現位置リストそのものを出力すればよい．しかし，クエリとタームが一致しない場合には，クエリが文書に出現したとしても，検索結果として出力できず，いわゆる検索漏れが起きてしまう．

一般にクエリとタームが一致するのを保証するのは困難である．例えば検

d_1 = {たぬきそばと，きつねうどんはきついね}
d_2 = {たぬきそばは，安い}
d_3 = {きつねの嫁入りを見た}

loc(たぬき) = [1, 2]
loc(きつね) = [1, 3]
loc(うどん) = [1]
⋮

図 7.1 転置索引の例

索対象文書が，“東京都特許許可局”であり，タームに区切ったとき，“[東京都][特許][許可局]”であったとする．このとき，“東京”で検索したとしても，タームの単位では“[東京都]”で記録されているので，この文書に東京が出現していることを見つけることができない．そのため，“東京都”という文字列が出現しているとき，“東京”と“東京都”の両方を記録しておくことも考えられるが，その場合，索引サイズは大きくなってしまう．

こうした検索漏れを防ぐため転置索引では，タームを細かい単位に設定しておき，クエリにタームが含まれる場合は，実際にそこに出現しているかどうかを実際のテキストを参照して調べるという方法がよく利用される．

また，転置索引は，タームをうまく設計すれば，索引サイズを小さく抑えることができる．例えばいわゆるストップワードと呼ばれる，文書中に大量に出現しているが，検索される可能性が低い単語をタームに含まなければ，索引サイズを大幅に小さくできる．

7.3.2 N-gram 索引

転置索引では，ターム単位と合わないクエリ文字列は検索が遅くなるだけでなく，出現しているのに検索結果に出てこないという検索漏れが発生する問題が発生する．こうした検索漏れは日本語など，単語境界が曖昧な言語で問題になるほかに，そもそも単語単位がわからないゲノム配列やログデータなどの文字列においては，どのようにタームを定義してよいのかが未知の場合も少なくなく，転置索引による検索は困難である．

d_1 = {たぬきそばと，きつねうどんはきつい ね}
d_2 = {たぬきそばは，安い}
d_3 = {きつねの嫁入りを見た}

⋮

loc =（きつ）= [(d_1, 7), (d_1, 14), (d_3, 1)]
loc =（つね）= [(d_1, 8), (d_3, 2)]
loc =（ねう）= [(d_1, 9)]
⋮

図 **7.2** N-gram 索引の例

N-gram 索引は長さ N の部分文字列すべてをタームと考え，それらの出現位置リストを記録する(図 7.2)．N-gram 索引の場合，あらゆるクエリ文字列を漏れなく検索できるという保証がある．なお，N-gram 索引は海外では Q-gram 索引と呼ばれる場合も多い．

例えば，長さ N=3 の場合の N-gram 索引を考えてみる．検索対象文字列が T=“*abracadabra*” であるとする．この場合のタームと，その出現位置のペアを並べたものは次のようになる．

$$(abr,0),(bra,1),(rac,2),(aca,3),(cad,4),(ada,5),(dab,6),(abr,7),(bra,8),$$
$$(ra\$,9),(a\$\$,10) \tag{7.1}$$

次に，同じタームの出現位置をまとめ，次のような N-gram 索引を得る．

$$(abr,[0,7]),(bra,[1,8]),(rac,[2]),(aca,3),(cad,4),(ada,5),(dab,6),$$
$$(ra\$,9),(a\$\$,10) \tag{7.2}$$

例えば，abr の出現位置は $[0,7]$ として求めることができる．

クエリ文字列がちょうど N 文字の場合は，対応するタームを探し，それに紐付く出現位置リストを返すだけでよい．問題となるのは，クエリ文字列が N 文字ではない場合である．はじめに，クエリ文字列が N 文字より長い場合を考える．まず，クエリ文字列を N 文字ごとに区切る，クエリ文字列の長さがちょうど N で割り切れない場合は最後の N 文字を含むように分割する．例

えばクエリ文字列が“*bracada*”であり，$N=3$ の場合は，“*bra*”, “*cad*”, “*ada*”と分割する．このとき，クエリ文字列が T 中で出現しているための必要十分条件は，クエリ文字列に含まれるタームすべてが，連続する位置で出現していることである．先ほどの例の場合，“*bra*”, “*cad*”, “*ada*”がそれぞれ，ちょうど 0, 3, 4 文字ずつずれた位置で連続して出現していればよい．よって，“*bra*”, “*cad*”, “*ada*”のタームに対応する出現位置リストを取得し，次に位置がクエリと同様にちょうどずれている集合を求めればよい．先ほどの例で対応する位置リストは，$(bra, [1, 8]), (cad, 4), (ada, 5)$ であり，これらの位置がちょうど 0, 3, 4 ずつずれている位置は $(bra, 1), (cad, 4), (ada, 5)$ なのでここに“*bracad*”が出現しているということがわかる．

この複数タームからなる場合に，連続して出現している位置を求める問題の計算量を考えてみる．2 つの位置集合リスト V, U が与えられたとし，一般性を失わずに $|V| \leq |U|$ であるとする．お互いの位置を小さい順にマージソートのように調べていく方法では，すべての要素を走査するので $\mathrm{O}(|V|+|U|)$ 時間必要である．それに対し，V のそれぞれの要素について，U 中に含まれるかどうかを二分探索で調べる方法では，$\mathrm{O}(|V| \log |U|)$ 時間かかる．この計算量は，二分探索で調べ終わった位置から次の値を調べれば減らすことができ，$\mathrm{O}(|V| \log |U|/|V|)$ 時間で調べることができる．位置集合リストのサイズに応じてこの 2 つの処理のどちらかを選べばよい．

次に，クエリ文字列が N より短い場合を考える．例えば，$N=3$ のとき，ab が出現している場所を調べるには，ab が接頭辞であるタームの結果をあわせればよい．こうして得られた出現位置には重複がなく，漏れもない．このとき，タームを辞書式順序で並べておけば，ある文字列が接頭辞であるタームを効率的に調べることができる．先ほどの例を辞書式順序で並び替えると次の通りである．

$$(a\$\$, 10), (abr, [0, 7]), (aca, 3), (ada, 5), (bra, [1, 8]), (cad, 4), (dab, 6), (ra\$, 9) \tag{7.3}$$

そして，a が接頭辞として出現しているタームは $(a\$\$, abr, aca, ada)$ の 3 つであり，これらの出現位置をすべて併せた $[10, 0, 7, 3, 5]$ が a の出現位置である．

タームは辞書式順序で並んでいるので，タームの数を T としたとき，クエリ文字列 a の出現区間は $O(\log T)$ 時間で求められる．

また，注意してほしいのは長さが N のタームの種類数は $O(\sigma^N)$ となるが，1 回以上出現したタームの種類数は，検索対象文字列の長さを n としたとき，高々 n 個である．よって，N-gram を用いた場合，このターム種類数は，$\min(n, \sigma^N)$ となる．

一方で，N-gram 索引を利用した検索では，処理時間を抑えることが難しい．例えば，クエリ文字列のヒット数が小さい場合でも，クエリ文字列に含まれるタームの出現回数が多い場合に計算量が大きくなってしまう．例えば，クエリ文字列が "*these*" の場合，"*these*" の出現回数が数回しかない場合でも，"*the*" と "*ese*" の出現回数が大きければ計算量は大きくなる．

7.3.3 差分リスト

転置索引と N-gram 索引はともに出現位置リストの集合を扱う．この出現位置リストを二進表現で格納した場合，サイズはもとのテキストと同じかそれ以上に大きくなってしまう．出現位置リストをいかに効率よく格納し，そのうえでさまざまな高速な操作を実現するかが重要となる．

出現位置リストを小さい順にソートし，隣合う要素の差分を求め，この差分を記録する方法が差分リストである．例えば，あるタームの出現回数が非常に多ければ，出現位置リスト中の各値の差分は小さく，差分リストに含まれる値には小さい値が多く出現するようになる．このとき，2.5 節で扱ったように，小さい値を短いビット数で表現すれば出現位置リストは短くできる．また，文書を似た文書が連続して出現するように並べることによって，出現位置リスト中で近い値が出現しやすいようにできる．例えば，ウェブ文書をクロールした場合，URL の辞書式順序で対応する文書を並び替えれば，似たような文書が近くに現れやすくなる．

差分リストでは，途中の値に直接アクセスできないため，複数のクエリが同時に出現している文書を探す場合や，N-gram 索引による複数タームが連続して出現している位置を探す場合に計算量が問題となる．そのため，差分リストではブロックごとに固めて圧縮し，各差分の先頭の文書番号を別に記憶してお

くことが多い．そして，これらの先頭の文章番号を参照すれば，アンド検索時に不必要なブロックを復元することなくスキップでき高速化が図れる．

以下では差分リストの符号化の代表的な方法である Var-Byte を挙げる．この他にもさまざまな特性をもった符号化法が提案されている[31]．

Var-Byte

差分リストの符号化に実用的に最も多く利用されているのが Var-Byte (Variable length Byte code)である．バイト単位で値を格納し，バイトの最上位ビットは後続バイトがあるかどうかを記録する．復元の際にはバイト単位で比較を行い，後続バイトが存在するのであれば 7 ビットシフトし，後続バイトを調べることを続ける．

この方法により，7 ビット未満で記録できる値は 1 バイト，14 ビット未満で記録できる値は 2 バイトで，といったように記録できる．小さい値が多く出現する差分リストでは Var-Byte を使うと多くが 1, 2 バイトで記録できる．すべての操作がバイト単位で行うことができるので，実装が容易であるという特徴がある．

7.3.4　ウェーブレット木を利用した転置索引

転置索引をウェーブレット木を用いて管理すれば，さまざまな操作を効率的に実現できる．各単語ごとの出現位置リストをつなげて作ったリスト D を作り，これに対するウェーブレット木を構築する．また，各単語に対応する出現位置リストが D 中のどの範囲に対応するのかを格納した完備辞書 B を，$D[i]$ が出現位置リストの開始位置ならば $B[i]=1$，そうでないならば $B[i]=0$ として作る．完備辞書 B を利用すれば i 番目のタームに対応する文書リストは $D[\mathbf{select}_1(B,i),\mathbf{select}_1(B,i+1))$ として取得できる．この表現方法はウェーブレット木を利用したグラフの表現と同じである．

これを利用してヒットした中で文書番号が小さい順，大きい順に列挙するのは，**rangemink**, **rangemaxk** を利用して求められる．また，**topk** を利用してヒット数が多い順に文書を列挙できる．また，2 つ以上のタームが両方出現するような文書リストを **intersect** を利用して高速に求めることが可能で

ある.

7.4 接尾辞配列による検索

任意の文字列を漏れなく高速に検索できるのが，接尾辞配列を索引として利用した検索である．接尾辞配列については 3.1 節で紹介した．今回は N-gram の拡張として接尾辞配列を考えてみよう．N-gram では長さ N の部分文字列ごとにその出現位置リストを記録した．それではこの N を長くしていき，検索対象テキストの長さまで伸ばした場合を考えてみよう．各タームが T の接尾辞 $T[i,n)$ に対応し，各タームは T 中の開始位置を指定すれば表現できるので，ターム自身を格納する必要はない．また，各タームはちょうど 1 回しか出現しないので各タームごとに集合位置リストを格納する必要はなく，ちょうど 1 つの位置のみを格納すればよい．これらの事実により接尾辞配列は長さ n の整数配列として効率的に格納できる．

接尾辞配列を用いて，任意のクエリ文字列 Q が文字列 T 中のどこに出現しているかは，N-gram 索引のときと同様に，Q を接頭辞として含む接尾辞配列の範囲 $SA[sp,ep)$ を探すことで実現される．この範囲においては，$sp \leq i < ep, T[SA[i], SA[i]+|Q|) = Q$ となる．対応する接尾辞の開始位置 sp，終了位置 ep は接尾辞配列上の二分探索で効率的に求めることができる．この出現範囲が求まればクエリ文字列の出現回数は $ep-sp$ で求められる．また，クエリ文字列の出現位置は $SA[sp,ep)$ で与えられる．

クエリ文字列の長さが m のとき，各二分探索のときの文字列比較に $\mathrm{O}(m)$ 時間かかり，$\mathrm{O}(\log n)$ 回の比較を行うので，合計で $\mathrm{O}(m \log n)$ 時間が必要である．この計算時間は出現回数に依存しないことに注意されたい．また必要な作業領域量は，アルファベットサイズが σ のとき，元テキストのサイズが $n \lg \sigma$ ビット，接尾辞配列のサイズが $n \lg n$ ビットである．なお，この計算量は次節で述べる FM-index を利用すれば，$\mathrm{O}(m \log \sigma)$ 時間に減らすことができる．

7.5　圧縮全文索引

全文索引のサイズはもとのテキストと同じか，それよりも大きくなる場合も少なくない．大量の文字列データを扱ううえで，索引自体を効率的に格納する必要もある．圧縮全文索引は，全文索引と簡潔データ構造，Burrows Wheeler 変換といった技術を組み合わせて構成され，索引自体を圧縮された状態のまま扱えるほかに，もとのテキストを必要としない特徴もある．

ここでは，代表的な圧縮全文索引である圧縮接尾辞配列と FM-index を紹介する．これらの変種や，他の圧縮全文索引については[6, 23]を参照してほしい．

7.5.1　圧縮接尾辞配列

接尾辞配列はサイズが $n \lg n$ ビットであり元テキストと比べても大きい問題があった．例えば $n{=}2^{32}$ であり，一般のテキストである $\sigma{=}256$ のとき，接尾辞配列は元テキストサイズの 4 倍のサイズになる．また，接尾辞配列は $[0, n)$ の整数の並び替えであるので，値の分布に偏りはなく，また局所的な値の偏りも見られないため，接尾辞配列はそのままでは圧縮できない．

圧縮接尾辞配列(Compressed Suffix Array, CSA)は，接尾辞配列を格納する代わりに，次のように定義される配列を格納する．

定義 7.2(ψ 配列)

$$\psi[i] = ISA[SA[i]{+}1] \tag{7.4}$$

□

この ψ 配列は，i 番目の順位にある接尾辞の次の位置から始まる接尾辞の順位を返す配列である．はじめに ψ 配列から SA の情報を復元できることを説明した後に，なぜ ψ 配列は圧縮できるのか，そして ψ 配列を用いて全文索引はどのように効率的に実現できるのかについてみていく．

ψ 配列を利用した SA の復元

はじめに接尾辞配列と逆接尾辞配列の役割を再度述べる．接尾辞配列 SA は接尾辞中の順位を引数として受け取りテキスト中の位置を返す配列であり，逆接尾辞配列 ISA はテキスト中の位置を引数として受け取り，接尾辞中の順位を返す配列である．定義より $SA[\psi[i]]=SA[i]+1$ が成り立つ．つまり，ψ 配列を利用すれば，順位 i の指す接尾辞の次の位置から始まる接尾辞の順位を取得できる．例えば順位が i である接尾辞がテキスト中の 4 番目の位置を指していれば，$\psi[i]$ はテキスト中の 5 番目からはじまる接尾辞の順位を示す．

この事実を再帰的に適用し，次を得る

$$\begin{aligned} SA[\psi^k[i]] &= SA[\psi^{k-1}[i]]+1 \\ &= SA[\psi^{k-2}[i]]+2 \\ &\vdots \\ &= SA[i]+k \end{aligned} \tag{7.5}$$

ただし，$\psi^1[i]=\psi[i]$, $\psi^k[i]=\psi[\psi^{k-1}[i]]$ である．

この関係を利用し，ψ 配列を利用して $SA[i]$ を復元する方法を説明する．l を適当な整数とし，SA の中で値が l の倍数，つまり $SA[i]=lk$, $0\leq k<n/l$ となる SA をサンプリングし，$SA_l[0,n/l)$ に格納する．また，$SA[i]=lk$ であれば $SA_b[i]=1$, そうでなければ $SA_b[i]=0$ となるようなビット列 $SA_b[0,n)$ を用意する．SA_b は完備辞書を利用して格納する．Algorithm 10 に，$SA[i]$ を ψ, SA_l, SA_b から復元する擬似コードを示す．

高々，l 回でサンプリングされている接尾辞配列に到達して接尾辞配列中の値を復元できる．このとき，任意の $\epsilon>0$ について $l=\log^{1+\epsilon} n$ とすれば SA_l のサイズを $o(n)$ とすることができる．

ψ 配列の圧縮

接尾辞配列とは違い ψ 配列は圧縮できる．はじめに ψ 配列の単調増加性について示す．

補題 7.3 配列 $T_F[0,n)$ を $T_F[i]=T[SA[i]]$ と定義する．つまり，$T_F[i]$ は i 番

Algorithm 10　$SA[i]$ を圧縮接尾辞配列から復元する

function SALookup(ψ, SA_l, SA_b, i)
　　$p \leftarrow i$
　　$k \leftarrow 0$
　　while $SA_b[p] \neq 1$ **do**
　　　　$p \leftarrow \psi[p]$
　　　　$k \leftarrow k+1$
　　end while
　　return $SA_l[\mathbf{rank}(SA_b, p)] - k$
end function

目の順位の接尾辞の先頭文字である．もし，$T_F[i]=T_F[i+1]$ なら，$\psi[i]<\psi[i+1]$ である．□

［証明］ 接尾辞を $S_i=T[i,n)$ と定義する．辞書式順序の定義より，$S_{SA[i]} <_{lec} S_{SA[i+1]}$ であるが，$T_F[i]=T_F[i+1]$ のときは 1 文字めは同じ文字であるため，接尾辞の 2 文字め以降の文字列で順位が決定されるはずであり $S_{SA[i]+1} <_{lec} S_{SA[i+1]+1}$ が成り立つ．$SA[i]+1$ から始まる接尾辞の順位は $SA[i+1]+1$ から始まる接尾辞の順位より小さく，$ISA[SA[i]+1]<ISA[SA[i+1]+1]$ である．これらの値はそれぞれ $\psi[i], \psi[i+1]$ である．■

よって，配列 ψ は T_F の文字が同じである範囲は単調増加列である．図 7.3 に，$T_F[i]=$“a” の間は ψ が単調増加列になっている例を示す．

T_F は $\{0,1,\cdots,\sigma-1\}$ が小さい順に並んでおり，それぞれの間の区間では ψ の単調増加列が存在し，それぞれの単調増加列の長さはその文字の出現回数である．各単調増加列は 0 から $n-1$ までの値をとり，それぞれの長さは各文字 $c \in \Sigma$ の出現回数 $f[c]$ に相当する．この単調増加列は完備辞書によって，各文字 c ごとに $f[c]\left(\lg e + \frac{\lg n}{f[c]}\right)$ ビットで格納できる．よって合計サイズは

$$\sum_c C[c](\lg e + \lg n / f[c]) = nH_0(T) + n \lg e \tag{7.6}$$

で格納できる．よって，ψ 配列は $nH_0(T)+n\lg e$ ビットで格納でき，定数時間で参照できることが示された．さらに，$\psi[i]$ 配列を対応する接尾辞の先頭 k 文字が同じもの同士のグループに分割すれば，ψ 配列は $nH_k(T)$ ビットで格

i	$\psi[i]$	BWT	Suffix
0	11	a	\$
1	0	r	$a\$$
2	6	d	$abra\$$
3	7	\$	$abracadabra\$$
4	8	r	$acadabra\$$
5	9	c	$adabra\$$
6	10	a	$bra\$$
7	3	a	$bracadabra\$$
8	5	a	$cadabra\$$
9	2	a	$dabra\$$
10	1	b	$ra\$$
11	4	b	$racadabra\$$

図 7.3 圧縮接尾辞配列において，ψ 配列は部分的に単調増加列となる．

納できる．圧縮接尾辞配列は全体で $nH_k(T)+o(n)$ ビットで格納できるが，接尾辞配列の各値へのアクセスには $O(\log^{1+\epsilon} n)$ 時間かかる．

圧縮接尾辞配列の検索

次に圧縮接尾辞配列を用いた検索をみてみよう．圧縮接尾辞配列を接尾辞配列の代わりに用いても，SA を復元せずに，ψ 配列のみを利用し，後方探索と呼ばれる方法で探索できる方法が知られている．

クエリ文字列 $Q[0,m)$ が与えられたとする．このアルゴリズムではクエリ文字列を後ろから順に調べ，範囲 $[sp,ep)$ を更新していく．$Q[i+1,m)$ まで処理が終ったとき，この範囲は $Q[i+1,m)$ を接頭辞として持つ接尾辞の範囲を示している．次の文字を $c{=}Q[i]$ としたとき，c に対応する値の範囲 $\psi[C[c],C[c+1])$ の中で，$sp{\le}\psi[i]{<}ep$ であるような範囲 $[sp',ep')$ を求める．この範囲は **rank** 操作で調べることができ $sp{:=}sp'$, $ep{:=}ep'$ と更新する．これを Q の

すべての文字に対し繰り返し適用すると，Q が接頭辞として出現する接頭辞の範囲 $[sp, ep)$ が得られる．

圧縮接尾辞配列の構築

圧縮接尾辞配列を構築するには，接尾辞配列を構築してから構築する方法と，文字列から直接構築する方法の 2 つが存在する．

接尾辞配列から構築する方法は Algorithm 2 で紹介した BWT の逆変換を途中まで行うことで求められる．この場合，もとの文字列と接尾辞配列，そして各文字の出現頻度を保持する配列 $C[0, \sigma)$ を利用して構築できる．このときに必要な作業領域量で支配的になるのが接尾辞配列の $n \lg n$ ビットである．

圧縮接尾辞配列を直接構築する場合は文字列 T の BWT を逐次的に直接構築する Algorithm 1 を応用し，ψ 配列を逐次的に構築する．しかし，BWT の逐次的な構築と同様に，実用的には接尾辞配列を介した構築よりは遅い．

7.5.2 FM-index

FM-index は現在最も高速で現実的な圧縮全文索引である．FM-index では BWT の LF-mapping の性質を利用し全文検索を行う．また，後述するさまざまなデータマイニングでも利用できる．

FM-index による検索

定理 3.8 では $T_B[0, n)$ に現れる各文字の順序と，$T_F[0, n)$ に現れる各文字の順序は一致する事実を示した．この事実を利用してクエリ文字列 $P[0, m)$ に対応する接尾辞配列の範囲 $SA[sp, ep)$ を求めることができる．

このアルゴリズムはクエリ文字列を後ろから順に操作し，範囲 $[sp, ep)$ を更新していく．はじめに $sp{:=}0$, $ep{:=}n$ と初期化する．また，$C[c]$ を T 中の文字 c より小さい文字の出現頻度とする．そして，$Q[i{+}1, m)$ まで処理が終ったとき，$SA[sp, ep)$ は $Q[i{+}1, m)$ を接頭辞として持つ接尾辞配列の範囲を示している．次のクエリ文字が $c{=}Q[i]$ であったとする．すると $Q[i, m)$ に対応する部分文字列は $Q[i{+}1, m)$ に対応する接尾辞 $SA[sp, ep)$ の範囲の中で文字 c が直前に出現しているはずである．別の言い方をすれば $Q[i{+}1, m)$ を接頭

Algorithm 11 BWT 配列 T_B を用いて文字列 $P[0, m-1]$ の接尾辞配列での一致範囲 $SA[sp, ep)$ の sp, ep を返す．配列 C は $C[c]$ に T_B 中の文字 c より小さい文字の出現数を格納する

```
function SearchFMIndex(T_B[0,n), C[0,σ), P[0,m))
    sp←0
    ep←n
    for i←1, m do
        c←P[m−i]
        sp←C[c]+rank_c(T_B, sp)
        ep←C[c]+rank_c(T_B, ep)
        if sp≥ep then
            return
        end if
    end for
    return (sp, ep)
end function
```

辞として持つ接尾辞配列の範囲が $SA[sp, ep)$ のときに，それらの直前の文字 $T_B[sp, ep)$ が c である場合が $Q[i, m)$ が出現している範囲に相当する．$T_B[sp, ep)$ の中で出現している c が T_B 中で何番めに出現しているかは，$\mathbf{rank}_c(T_B, sp)$，$\mathbf{rank}_c(T_B, ep)$ として求められる．ここで各文字の順序が T_B と T_F で変わらないことを利用して，sp, ep を次のように更新することで $SA[sp, ep)$ は $Q[i, m)$ に対応する接尾辞配列の範囲となる．

$$sp := \mathbf{rank}_c(T_B, sp) + C[c] \tag{7.7}$$

$$ep := \mathbf{rank}_c(T_B, ep) + C[c] \tag{7.8}$$

Algorithm 11 に FM-index を用いた全文検索の例を示した．クエリ文字列 $P[0, m-1]$ が出現している接尾辞配列の範囲を返す．$ep-sp$ が出現回数であり，$SA[sp, ep-1]$ が出現している位置である．

つまり，T_B 中の文字列の数を数えるだけで全文索引が行えるのである．この計算量は，各 **rank** 操作はウェーブレット木を利用すれば $O(\log\sigma)$ 時間で求められるので，任意のクエリ文字列 $P[0, m-1]$ の出現回数を $O(m\log\sigma)$ 時

間で求められることが示された．また，ここから出現位置を復元する際は圧縮接尾辞配列で紹介した接尾辞配列のサンプリングを利用すればよい．この際には ψ 配列の代わりに $\mathbf{rank}_c(T_B, c)+C[c]=ISA[SA[i]-1]$ であることを利用して，サンプリングされた接尾辞配列から復元する．

FM-index の効率的な格納

さて，BWT 配列はウェーブレット木でどのように効率的に格納できるのだろうか．BWT 配列は元の文字列の並び替えであったことから，各文字の出現頻度は変わらないので，$H_0(T)=H_0(T_B)$ が成り立つ．よって，BWT 配列のウェーブレット木のアルファベット木を T の頻度から求められたハフマン木を利用すれば，サイズを $nH_0(T)$ ビットにできる．また，クエリ文字列の各文字の出現頻度が T と一致するならクエリ文字列の計算量の期待値は $\mathrm{O}(mH_0)$ である．この場合の最悪計算量はハフマン木の高さを $\mathrm{O}(\log\sigma)$ に抑える方法を利用すれば，$\mathrm{O}(m\log\sigma)$ 時間にできる．

しかし，BWT 配列は第 6 章でも扱ったように，同じような文字が近くで出現しやすいという特徴があり，それを利用してさらに圧縮できるはずである．BWT 配列をその対応する接尾辞の長さ k の接頭辞が同じもの同士ごとに分割し，それぞれを 0 次経験エントロピーのサイズで格納すれば，合計サイズは $nH_k(T)$ になる．正確には次のように述べることができる．

補題 7.4 任意の $k\geq 0$ において，$L_1L_2\cdots L_l=BWT(T)$, $l\leq\sigma^k$ であり，

$$\sum_{i=1}^{l}|L_i|H_0(L_i) = nH_k(T) \tag{7.9}$$

となるような，分割 L が存在する．これを Compression Boosting と呼ぶ． □

しかし，最大 σ^k 個のグループに分割し，それぞれを別々のハフマン木を利用してウェーブレット木で格納した場合，計算量および作業領域量のオーバーヘッドは大きくなる．

この問題に対し，固定長ブロックごとに分割しても同じように $nH_k(T)$ ビットを達成できることが示された[15]．はじめに，$H(x,y)$ を x 個の 0 と y 個の 1 を含むようなビット列 B の $|B|H_0(B)$ とする．また，$n_c(T)$ を文字列 T 中の文字 c の出現回数とする．

補題 7.5　任意の文字列 X, Y に対し次が成り立つ.

$$\begin{aligned} 0 &\le |XY|H_0(XY) - |X|H_0(X) - |Y|H_0(Y) \\ &= H(|X|,|Y|) - \sum_{c\in\{0,1\}} H(n_c(X), n_c(Y)) \\ &\le H(|X|,|Y|) \le |XY| \end{aligned} \tag{7.10}$$

□

［証明］　最初の不等式はギブスの不等式より，最後の 2 つの不等式は，エントロピーの定義より明らかである．等式部分については，両辺を展開すれば確認される．■

この補題 7.5 からは，2 つの文字列をつなげることにより，全体のエントロピーは下がらず，また増加分は 1 文字あたり高々 1 ビットであることがいえる．

この補題を利用し，BWT 文字列を k 文字の文脈に応じた分割から，固定長の分割に変更したときのエントロピーの増加分を計算してみよう．

補題 7.6　$X_1X_2\cdots X_l=X$ を文字列 X の l 個のブロックへの任意の分割とし，$X_1^bX_2^b\cdots X_m^b=X$ を X の最大長が b であるブロックの分割であるとする．このとき，次が成り立つ．

$$\sum_{i=1}^{m} |X_i^b|H_0(X_i^b) \le \sum_{i=1}^{l} |X_i|H_0(X_i) + (l-1)b \tag{7.11}$$

□

［証明］　はじめに，$X_1X_2\cdots X_l=X$ の分割が与えられ，それに，$X_1^bX_2^b\cdots X_m^b=X$ に対応する分割を加えた後に最初の分割を削除した場合を考える．補題 7.5 より，分割点を加えることによりエントロピーは増加せず，また分割点の削除により増えるエントロピーは分割点の削除 1 つあたり高々 b ビットである．■

この補題を利用し，次の結果が得られる．

定理 7.7　任意の $\varepsilon>0$ について，$k\le((1-\varepsilon)\log_\sigma n)-1$ のとき，FM-index は $nH_k(T)+o(n)\lg\sigma$ ビットで格納できる．□

［証明］　Compression Boosting に基づく分割から，固定長のブロックの分

割へ変更した場合のサイズ増加分は $b=\sigma(\log n)^2$ と設定したとき，補題 7.6 より，$\sigma^k b=\sigma^{k+1}(\log n)^2=n^{1-\epsilon}(\log n)^2=\mathrm{o}(n)$ ビットである． ■

BWT を単に固定長ブロック一定長に分割し，それぞれを 0 次経験エントロピーで格納すれば全体のサイズを $nH_k(T)$ に近いサイズで記録できることがわかる．また，全体の **rank** の操作を行うためにブロックごとに各文字の **rank** の結果を保存する．このサイズは $(n/b)\sigma\lg n=n/\lg n=\mathrm{o}(n)$ である．

7.6　キーワード集合の管理

文書集合を扱うデータ構造と並んで重要なのが，キーワード集合を扱うデータ構造である．

定義 7.8（キーワード辞書の操作）　キーワード集合 $K=\{k_1,k_2,\cdots,k_m\}$ が与えられたとき，次の 2 つの操作を定義する．

- **membership**(q):　クエリ文字列 q がキー集合に含まれていたら `True`，含まれていなければ `False` を返す．
- **prefix**(q): クエリ文字列 q の接頭辞と最長一致するキーワードを返す．

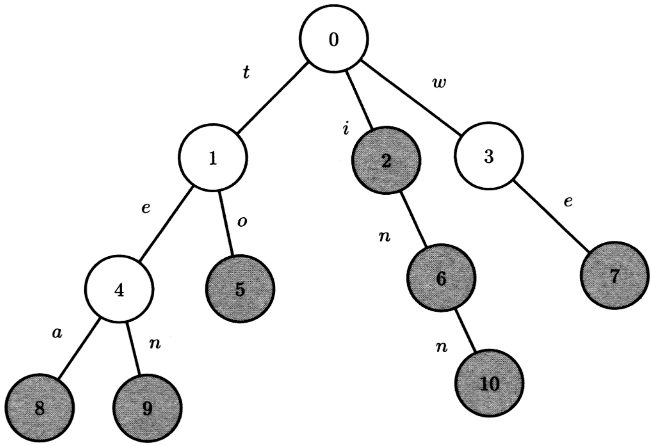
□

これらの操作を高速に処理しつつ，キーワード集合を小さく保存するデータ構造を紹介する．

7.6.1　Trie

キー集合を扱う代表的なデータ構造が Trie である．Trie は木構造からなる．各枝には文字が付随し，根から葉へのパス上の枝をつないだ文字がキーに対応する．また，キーが他のキーの接頭辞である場合は，キーは途中の節点で終了するので各節点 t は，そこでキーが終了したかどうかを示すビット $T[t]\in\{0,1\}$ を格納する．図 7.4 の上に Trie の例を挙げる．網がかかっている節点は，キーが終了したことを意味している．

Trie を利用して各操作は次のように実現できる．**membership** はクエリ文字列にしたがって根から各節点をたどって行く．節点（葉）t までたどり着いたとき $T[t]=1$ ならば `True` を返し，途中でたどれなくなった場合や，たど



$\{$"to", "tea", "ten", "i", "in", "inn", "we"$\}$

	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
B	0	0	0	1	0	0	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1
T	0	0	1	0	0	1	1	1	1	1	1										
E	t	i	w	e	o	n	e	a	n	n											

図 7.4 上はキー集合 $\{$"to", "tea", "ten", "i", "in", "inn", "we"$\}$ に対する Trie の例．下はその Trie の簡潔表現．配列 B は Trie の木構造の LOUDS 表現であり，配列 T は各節点ごとに，そこで終了するキーが存在するかを示し，配列 E は枝に付随する文字を記録する．配列 E は B 中の 0 に対応する．

れたとしても $T[t]=0$ ならば **False** を返す．**prefix** 操作の場合も同様である．この場合は最後に $T[t]=1$ であった節点 t に対応するキーを返せばよい．例えば **membership**(ten) は節点 1, 4, 9 をたどってそこがキーの終了であるので **True** を返す．

Trie はそのまま格納した場合作業領域量が大きくなる問題がある．例えば Trie 中のノードの数が n 個のとき，木のポインタ表現は $\lg n$ ビット必要なので，全体で $O(n \log n)$ ビット必要である．このままでは非常に大規模なキー集合を扱うことは困難である．

この Trie を効率的に表現する．まず，木構造については，4.3 節で扱った簡

潔木である LOUDS 表現を利用する．簡潔木は節点数が n のとき，高々 $2n+o(n)$ ビットで木を表現し，木上での操作を実現する手法であった．

Trie の枝の長さの総長を l とする．これはキーを辞書式順序で並べたときに各キーの直前のキーと異なる長さの総長と一致する．各枝に付随する文字をすべてつなげた配列を $E[0,l)$ とする，これは $l \lg \sigma$ ビットで表現できる．LOUDS 表現での 0 と配列 E での対応は E での **rank**, **select** 操作により付けられる．これらを合わせると，Trie は LOUDS 表現による簡潔木によって，高々 $2l+l \lg \sigma+o(l \lg \sigma)$ ビットで表現できる．

各内部節点で子の数は高々 σ 個あるが，二分探索で求めれば，長さ m のクエリ文字列が出現しているかどうかは $O(m \log \sigma)$ 時間で調べることができる．

図 7.4 下に Trie の簡潔表現の例をあげる．先ほどの例の Trie 木を 3 つの配列 B, T, E によって表現している．

7.6.2 XBW

さらに効率よく Trie 木を表現する方法が XBW[7] である．各節点 t について，t から根までの枝に付随するラベルをつなげた文字列を t の接頭辞パスと呼び $P(t)$ で表す．このとき，接頭辞パスの最後には他の場所では出現せず最小である特殊文字 \$ をつける．また，節点 t からそれの子に向かう枝の文字を t の分岐文字と呼び $X(t)$ で表す．また，その節点で終了するキーが存在した場合，分岐文字に特殊文字 # を含めることで，これを表現する．1 つの節点に対応する分岐文字 $X(t)$ はアルファベットの部分集合である．

Trie 中の全節点をその接頭辞パスをキーとし，辞書式順序でソートして並び変え，その順序が $t_1, t_2, t_3, \cdots, t_l$ だったとする．このとき分岐文字をその順序にしたがって並び替えた文字列 $X(t_1)X(t_2)\cdots X(t_l)$ が XBW である．XBW は BWT の自然な拡張であり，BWT は葉から根に向かう一本鎖の木に対する XBW だと考えられる．これをみるために文字列 T の各文字を，一本鎖の木の葉から根に向かって順に対応付けたとする．このとき，各節点の接頭辞パスと，T 中の接尾辞は一致し，また分岐文字が BWT の各文字に対応する．

図 7.5 に先ほどの Trie と同じキー集合に対する XBW の例を示す．各分岐

{"*to*","*tea*","*ten*","*i*","*in*","*inn*","*we*"}

i	X 分岐文字	P 接頭辞パス
0	t, i, w	$
1	#	$aet$$
2	a, n	$et$$
3	#	$ew$$
4	n, #	$i$$
5	#	$net$$
6	n, #	$ni$$
7	#	$nni$$
8	#	$ot$$
9	e, o	$t$$
10	e	$w$$

図 **7.5**　XBW

文字が接頭辞パスをキーとしてソートされており，この分岐文字を保持する．

XBW 上でも BWT と同じように，LF-mapping の関係が成り立つ．つまり，各文字について，分岐文字 $X[0,i)$ 中の順位と，接頭辞パス中の最初の文字の順位は変わらない．$\mathbf{rank}_c(X,i)$ を分岐文字中 $X[0,i)$ 中で文字 c が出現した回数とする．また $C[c]$ を分岐文字集合の中で文字 c より小さい文字が出現した回数とする．このとき，XBW 中の i 番めの節点の子で文字 c に対応する子が存在する場合，$\mathbf{rank}_c(X,i)+C[c]$ がその子の XBW 中の節点の位置を示す．例えば，図 7.5 の i=4 の分岐文字 n は分岐文字中では 2 番めの出現であるが(1 番めは i=2)，これは接頭辞パスの先頭文字で 2 番めの n の出現 i=6 に対応する．

これよりキーを使って **membership** は次のように計算される．根に対応する節点 i=0 から始め，各節点において $c \in X(i)$ であるかどうかを調べる．もし，そうであれば $\mathbf{rank}_c(X,i)+C[c]$ を用いて子に移動し，そうでなければ，その時点で `False` を返す．そして最後にたどりついた節点 i で分岐文字 $X(i)$

の中にキーの終了を示す文字 $\#\in X(i)$ があるかどうかを調べ，あるなら `True` を返し，そうでないなら `False` を返す．

これを実現しようとしたときの問題は分岐文字を格納する X は各位置で複数文字を格納するため，**rank** 操作および **access** 操作はそのままウェーブレット木を用いて実現できないことである．

これを解決する 1 つめの方法は，X の各文字を順に並べてつなげた文字列を作り，また各分岐文字がどこから開始しているのかを示す完備辞書を構築する方法である．これより，先ほどの **rank**, **access** 操作は完備辞書と組み合わせて実現できる．例えば，**rank** 操作の実現にウェーブレット木を利用した場合，XBW を用いた文字列クエリは $O(|q|\log\sigma)$ 時間で求めることができる．必要な作業領域量は，全枝長が l のとき，$l\lg\sigma+2l$ ビットである．

もう 1 つの方法は，各文字 $c\in\Sigma$ ごとに i 番めの節点に出現したら 1，そうでないならば 0 であるようなビット配列を作りそれに対する完備辞書を作る．この場合，アルファベット数に依存しない定数時間で **access**, **rank** 操作を行い節点をたどることができる．しかしこの場合，節点における分岐文字を調べる操作は各 $c\in\Sigma$ について調べなければならないので，最悪 $O(\sigma)$ 時間必要である．

XBW は BWT と同じように分岐文字は圧縮しやすいように並び替えられているという特徴がある．同じような接頭辞パスの子は同じ文字が出現しやすいので先ほどの作業領域量は減らすことができる．

7.6.3　接頭接尾索引

これまで接頭辞が p であるキーを列挙する問題を主に扱ってきた．これを利用し，接尾辞が s であるキーを列挙する問題もキーを逆順に格納し，クエリ文字列を逆順に調べれば容易に可能である．ここではこの問題を拡張し，接頭辞が p であり，かつ接尾辞が s であるようなキーをすべて列挙する **prefixsuffix**(p,s) を考える．

キーワード集合は辞書式順序でソートされ $\{k_0,k_1,\cdots,k_{m-1}\}$ であったとする．これらキーワードを $T=\$k_0\$k_1\$\cdots\$k_{m-1}\$\#$ とつなげた文字列を考える．ただし $\$,\#$ はそれぞれキーワード中に出現するどの文字よりも小さいとす

Algorithm 12 BWT 配列 T_B を用いて接頭辞が $P[0, m_1)$，接尾辞が $S[0, m_2)$ であるキーワードを返す

```
function SearchPrefixSuffix(T_B[0,n), P[0,m_1), S[0,m_2))
    sp←0
    ep←n
    Q←S$P
    m←|Q|
    for i←1,m do
        c←Q[m−i]
        sp←C(c)+rank_c(T_B, sp)
        ep←C(c)+rank_c(T_B, ep)
        if c=$ then
            sp←sp+1
            ep←ep+1
        end if
        if sp≥ep then
            return
        end if
    end for
    return (sp, ep)
end function
```

る．文字列 T に対する BWT である T_B を構築し，さらに T_B から FM-index を構築する．

この FM-index による全文索引を利用すれば接頭辞が p であるキーワードを検索するには，$\$p$ を全文検索すればよく，接尾辞探索は $s\$$ を探索すればよい．しかし，p が接頭辞であり，s が接尾辞であるようなキーを全文検索で求めることは自明ではない．

この問題に対しては簡単な工夫で答えが求められることが知られている[9]．はじめに，理解しやすいようにキーワード間の境界文字を次のように番号付けをしたとする $T=\$_0k_0\$_1k_1\cdots\$_{m-1}k_{m-1}\$_m\#$．接頭辞 $\$p$ が接尾辞配列中で出現した範囲を $[sp, ep)$ とする．このとき，$\$$ が最小であることから，これら

のキーは $k_{sp}, k_{sp+1}, \cdots, k_{ep-1}$ に対応する．次に接尾辞として s を持つ文字列を探す．文字列 k_{sp} の末尾は $\$_{sp+1}$ であるが，この位置は $SA[sp+1]$ として求められる．同様に他の末尾はすべて現在の接尾辞配列中の位置に $+1$ した位置が末尾の位置に対応する．よって，はじめに p を後方探索し，そのあとにキーワード境界を超えるときに接尾辞の範囲を $sp{:=}sp+1, ep{:=}ep+1$ と更新したうえで，続けて s を後方探索すれば p を接頭辞かつ s を接尾辞として持つキーワードを探すことができる．この $[sp, ep)$ を求める時間は接尾辞と接頭辞を合わせた長さが m のとき $\mathrm{O}(m \lg \sigma)$ 時間である．Algorithm 12 にこの擬似コードを示す．

7.7　アルファベットサンプリング

最後に，この章で紹介したどの手法でも利用可能なアルファベットサンプリングを紹介する．

アルファベット集合 Σ の中から，いくつかアルファベットを選び，これを Σ_k とする．次に検索対象の文字列 $T[0,n)$ から，Σ_k に属する文字だけからなる文字列 $T_k[0,n')$ を構築する．このとき，$n' \leq n$ である．また文字列 T の各文字 $T[i]$ について Σ_k 中の文字であれば $B[i]=1$, そうでない場合は $B[i]=0$ であるようなビット列 B を作り，それに対する完備辞書を構築する．これより T 中の位置と T_k 中の位置が B の **rank**, **select** 操作で求められる．例えば，$T_k[p]$ は，$T[\mathbf{select}_1(i,p)]$ に対応する．

次にクエリ文字列 Q に対し，T と同じように Σ_k に属する文字だけからなる文字列 Q_k を構築する．クエリ文字列 Q が T で出現している場所を求める場合，T_k 中に Q_k が出現している箇所があるかどうかを調べる．もし，T 中に Q が出現している箇所があれば，それは T_k 中でも Q_k は出現しているはずである．

例として，$T=$“*abracadabra*” に対し，$\Sigma_k=\{b,c,r\}$ としたとき $T_k=$“*brcbr*” であり，$B=01101001010$ である．$Q=$“*abra*” の場合，$Q_k=$“*br*” となる．そして，T_k 中で Q_k が出現しているのは，$\{0,3\}$ である．これらの T_k 中の位置を **select** 操作を利用して T 中の位置 $\{0,7\}$ に戻し，次にこれらの出現位置に

ついて，実際もとの文字が出現しているかを調べる．うまくアルファベットを選ぶことで T_k は非常に小さくできる．また，この手法は逐次検索でも全文索引でも同様の考えで利用できる．

この方法で問題となるのは，アルファベット種類数が少ない場合(例えば $\sigma=4$ である DNA 配列など)だと，クエリ文字列がサンプリングの結果，空文字となってしまう場合である．こうした場合，すべての位置が候補位置となってしまう．その場合はもとの文字列 T も保存しておき，それに対し従来の逐次検索を行うことが考えられる．

アルファベットサンプリングはクエリ文字列の各文字の出現確率に大きな偏りがある場合は特に有効な方法である．例えば，もとの文字列には頻出するが，クエリ文字列には滅多に出現しないような文字は Σ_k に含めないようにすることで検索を大幅に高速化できる．

8

テキストマイニングのためのデータ構造

この章では，これまで紹介してきた文字列解析のツールを利用し，一見不可能と思われるような大規模かつ深いテキストマイニングを実現するのが目標である．一般にテキストマイニングにおいては，テキストから単語やフレーズを抽出したうえで，これらの出現頻度，相関，変化などを解析する．本章では，こうした解析の基礎となる技術を説明し，非常に巨大なテキストに対しても高速に解析ができることを示す．

8.1　接尾辞木と極大部分文字列

これまで紹介してきた接尾辞配列や BWT などの全文索引を用いて，文字列の頻度やその出現位置を求めることは高速に行える．しかし，例えば，“2 つの部分文字列に共通して出現する最長の部分文字列を求める” といった高度な文字列処理を効率的に行うには接尾辞木の考え方が必要となる．

8.1.1　接尾辞木

接尾辞木は，文字列中に含まれるさまざまな情報を効率的に格納し，文字列解析を効率的に実現するデータ構造である．

接尾辞木の説明の前にパトリシア木について説明する．パトリシア木のもとになるのは 7.6.1 節であつかった Trie である．Trie は文字列集合が与えられたとき，根から葉へのパス上の枝の文字をつなげると各文字列に対応するような木であった．この Trie から，子が 1 つだけの節点は子とマージして，その節点と子の間の枝の文字をその節点への枝の文字と連結し，各枝に長さ 1 以上の文字列が付随するようにする．このようにして，すべての節点が 2 つ以上の子を持つようにして構成された木をパトリシア木と呼ぶ．簡単にわかる事実として，n 個の文字列からなるパトリシア木の内部節点の数は高々 $n-1$ 個である．なぜなら葉の数はキーの数と同じ n 個であり，各節点では必ず 2 つ以上の子が存在するので，節点 1 つごとにキーが 1 つずつまとめられるからである．

文字列 T の接尾辞集合から構成されるパトリシア木を考える．テキスト長が n のとき，接尾辞の数は n であり，内部節点の数は高々 $n-1$ 個，葉の数は n 個である．しかし各枝に付随する文字列の合計長は，最悪 $O(n^2)$ となりうる．そのため，枝の文字をそのまま格納するのではなく T 中の開始位置と枝長のペアを保存する．葉の場合は接尾辞に対応するので開始位置のみを記録すればよい．以上をまとめると，枝に付随する部分文字列は高々 $O(n)$ 個の整数と文字列 T 自身により表現できる．このような木を T の接尾辞木と呼ぶ．また葉に付随する整数をすべてつなげたものが接尾辞配列に対応する．

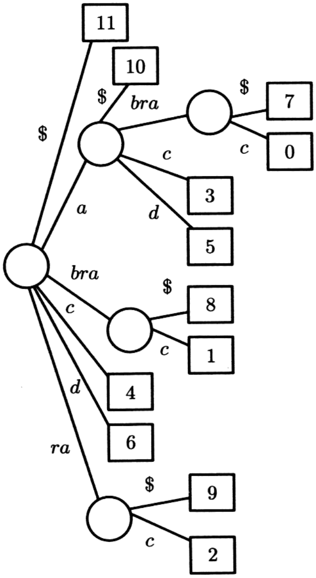


図 8.1 接尾辞木の例．ただし葉へ向うパスに付随する文字は最初の 1 文字めのみを示している．

図 8.1 に T=“*abracadabra$*” に対する接尾辞木の例を挙げた．接尾辞木の重要な性質として，文字列 T の部分文字列 Q に対応する T の接尾辞木の位置は根から枝の文字にしたがってたどり着いた位置であり，Q の T 中の出現位置は，その位置の子孫である葉の位置集合に対応する．このように文字列 T に含まれるすべての部分文字列は，T の接尾辞木中の位置として表現できる．

文字列 $T[0,n)$ に対する接尾辞木は，アルファベットサイズによらずに次のように $\mathrm{O}(n)$ 時間で構築できる．はじめに，接尾辞配列 $SA[0,n)$ を構築する．次に最長共通接頭辞配列 $LCA[0,n)$ を構築する．これは $LCA[i]$ が $S_{SA[i]}$ と $S_{SA[i+1]}$ の共通な接頭辞の長さを保持するような配列である，ただし $LCA[n-1]=0$ である．この $LCA[0,n)$ の構築には，定義通り比較すると最悪 $\mathrm{O}(n^2)$ 回の文字比較が必要であるが，$\mathrm{O}(n)$ 時間で構築できる方法が知られている[16]．次に LCA を用いて接尾辞木の節点を順に列挙する．これは $\mathrm{O}(n)$ 時間で調べることができる[16]．

接尾辞木のサイズは接尾辞配列と比べると大きい．接尾辞木では各節点ごとに，枝に付随する文字列の開始位置，隣接する節点へのポインタを保持する必要がある．接尾辞配列のサイズは $n \lg n$ ビット，BWT のサイズは $n \lg \sigma$ ビットであったのに対し，接尾辞木のサイズは少なくとも $4n \lg n$ ビットと大きい．この問題を解決するために簡潔データ構造を用いて接尾辞木をコンパクトに表現する方法がいくつか提案されている．例えば，木構造を簡潔木を用いて表現し，接尾辞配列は圧縮接尾辞配列，もしくは FM-index を利用して表現することで，全体サイズを $6n+|CSA|+\mathrm{o}(n \lg \sigma)$ ビットまで抑えることができる．ただし，$|CSA|$ は T を圧縮接尾辞配列を用いて表現したときのサイズである．また，疎化接尾辞木と呼ばれるデータ構造では，$\mathrm{o}(n)+|CSA|$ ビットで接尾辞木を表現できる[29]．

8.1.2　接尾辞木を利用した解析

接尾辞木を利用して，1.3 節で扱った問題をどのように解けるかを考えてみる．この問題は 2 つの文字列 S, T が与えられたとき，S のみに出現し，T には出現しない部分文字列をすべて漏れなく重複なく列挙せよという問題であった．文字列 X に対し X 中に出現するすべての部分文字列からなる集合を $G(X)$ とする．例えば，S=“*kyoto*”，T=“*tokyo*” の場合，求めたい部分文字列集合 $G(S)-G(T)$ は，

$$G(S) = \{k, y, o, t, ky, yo, ot, to, kyo, yot, oto, kyot, yoto, kyoto\} \tag{8.1}$$

$$G(T) = \{t, o, k, y, to, ok, ky, yo, tok, oky, kyo, toky, okyo, tokyo\} \tag{8.2}$$

$$G(S)-G(T) = \{ot, yot, oto, kyot, yoto, kyoto\} \tag{8.3}$$

となる．

このように S と T に出現するすべての部分文字列 $G(S)$, $G(T)$ を明示的に列挙し比較した場合，計算量が問題となる．長さ n の文字列中の部分文字列の数は $n(n+1)/2$ 個である．なので $G(S)$ と $G(T)$ をすべて比較した場合，文字列間の比較が定数時間でできたとしても $\mathrm{O}(|S|^2+|T|^2)$ 時間必要である．

この手法は文字列の長さが数万程度で破綻してしまう．しかしこれから示す方法で接尾辞木を利用すれば長さが数十億からなる文字列も扱えるようにな

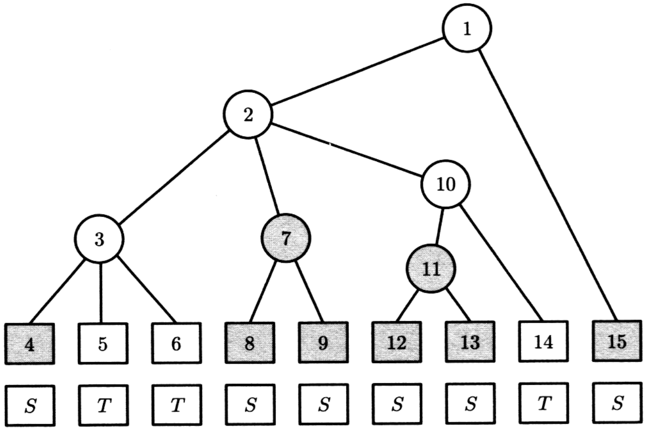


図 8.2 接尾辞木を利用した S のみに出現する部分文字列（アミかけの節点）

る．S と T をつなげた文字列 $U{=}S\$T$ に対する接尾辞木を考える，ただし \$ は S, T のどちらにも出現しない特殊文字とする．次に接尾辞木の各節点ごとにそれに対応する部分文字列が S 由来だけからなるか，そうでないかを葉から根に向かって順にしらべていく．はじめに接尾辞木の葉それぞれに対し，それが S 由来ならばその葉をマークする．次に，接尾辞木の各節点を順にたどって行き，ある節点 t の子がすべてマークされているならば，t の部分文字列も S 由来であるとわかるので，t もマークする．しかし，1 つでも子にマークされていない節点が存在する場合は，T にその部分文字列が出現していることになるのでマークをつけない．マークがつけられた節点は S 由来の節点であるので，その節点に対応する部分文字列も S のみに出現する．こうして，S だけに存在する部分文字列をすべて漏れなく重複なく列挙できる．例えば，図 8.2 の例では，番号 3 が付けられている節点では，S と T の両方の文字列で出現しているので，報告せず終了する．それに対し，内部節点 $7, 11$ はそれぞれ S のみで出現しているのでマークし報告する．

この方法による計算量を以下に見てみよう．接尾辞配列を構築するのに $O(|T|{+}|S|)$ 時間，また節点の数は葉を含めても高々 $2(|T|{+}|S|){-}1$ 個であるので節点を列挙するのも $O(|T|{+}|S|)$ 時間で行うことができる．必要作業領域

量は SA と接尾辞木の内部節点列挙で必要な深さ配列 LCA であり，全体で $2n\lg n$ ビットである．ここで注意して欲しいのは報告するすべての部分文字列の数は $O(|S|^2)$ となりうるので，報告する時間だけで $O(|S|^2)$ かかってしまうが，同じ枝に所属する部分文字列をまとめて報告するようにすれば，高々 $|S|-1$ 個の部分文字列情報を報告するだけで済む．例えば，節点 t を報告する場合，t の深さを d，t の親の深さを $d_p<d$ としたとき，$d_p+1\leq l\leq d$ の長さの部分文字列が対応すると報告すればよい．

この問題を，複数の文字列に拡張した場合でも解けるように拡張するのは容易である．先ほどと同じように複数の文字列を特殊文字を挟んで１つの文字列を作り，それに対する接尾辞木を作り各節点をたどっていけばよい．例えば，文字列集合 $T_1, T_2, \cdots, T_m$ の中から，k 個以下の文字列にしか出現していない部分文字列をすべて列挙するといったこともできる．いずれの場合も $n=\sum_i |T_i|$ のとき，$O(n)$ 時間で列挙可能であり，必要な作業領域量は $O(n\lg n)$ ビットである．

この例から示唆されるのは，文字列 T 中に出現する部分文字列の総数は $O(|T|^2)$ 個であるが，それらの統計量は高々 $2|T|-1$ 個の接尾辞木中の節点の情報にまとめられるということである．次の節で実際にどのような統計量が効率的に求められるかを見てみる．

8.1.3 極大部分文字列

ここでは，文字列中に含まれる部分文字列の中の代表である極大部分文字列を紹介する．先ほどみてきたように，文字列中に含まれる部分文字列は出現位置が同じもの同士でグループ化できる．

文字列 T が与えられたとき，２つの部分文字列 Q_1, Q_2 が $\mathbf{locate}(T, Q_1)=\mathbf{locate}(T, Q_2)$ を満たす，つまり同じ場所でしか出現していないような文字列はどのような場合かを考える．接尾辞木の上で考えれば，Q_1, Q_2 が同じ枝に対応する部分文字列である場合にこれは成り立つ．なぜなら同じ枝に対応する部分文字列については，その出現位置を示す子孫の葉の集合も一致するからである．例えば，図 8.1 の節点 “a” と “$abra$” の間の枝に対応する部分文字列 “ab”, “abr”, “$abra$” の文字列中での出現位置は一致し，$\mathbf{locate}(T, ab)=$

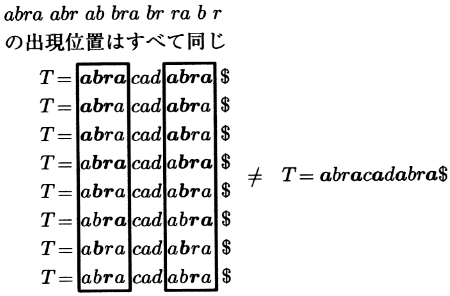


図 8.3 極大部分文字列

locate(T, abr)=**locate**$(T, abra)$=$\{7, 0\}$ である．

集合 $P=\{p_1, p_2, \cdots\}$ と整数 $c \in Z$ が与えられたとき，$P-c=\{p_1-c, p_2-c, \cdots\}$ と定義する．2 つの部分文字列 Q_1, Q_2 について，$Q_1=\alpha Q_2 \beta$ であったとする，ただし α, β は空文字列を含む任意の部分文字列である．このとき，**locate**$(T, Q_1)-|\alpha|$=**locate**(Q_2) である場合，$Q_1 =_P Q_2$ と定義する．直感的には，Q_1 と Q_2 の T 中の出現位置は $|\alpha|$ だけずれたうえですべて同じであり，Q_2 が出現しているところでは必ず Q_1 も出現していることを意味する．

この $=_P$ 関係が成り立つ部分文字列を同じグループだとみなした場合，T 中に出現するすべての部分文字列はいくつかのグループに分割される．このグループ内で一番長い部分文字列を極大部分文字列と呼ぶ．そして極大部分文字列の中で T 中の出現回数が 2 回以上であるものを M_T で表す．例えば，T=“$abracadabra\$$” の場合，$M_T=\{a, abra\}$ であり，これらはそれぞれ $\{a\}$，$\{ab, abr, abra, b, br, bra, r, ra\}$ のグループの中で一番長い部分文字列に相当する．図 8.3 にこれらの例を示す．

ある部分文字列が極大部分文字列である必要十分条件は，その部分文字列が接尾辞木中の節点に対応し，さらにその節点が対応する BWT の文字が 2 種類以上から構成されることである．このことより極大部分文字列の数は高々 $2n-1$ 個であることが示せる．先ほどの例では，接尾辞以外では a, $abra$ が極大部分文字列である．

節点が対応する BWT の文字集合が 2 種類以上からなっているかは定数時

間で判定できる．BWT 配列 T_B に対し $T_B[i]=T_B[i-1]$ のとき $B[i]=0$，そうでないとき $B[i]=1$ となるような完備辞書を構築すれば，$B[sp, ep)$ が 2 種類以上の文字からなるかは $\mathbf{rank}_1(B, ep)-\mathbf{rank}_1(B, sp+1)>0$ で判定できる．

接尾辞木の節点の列挙は $O(n)$ 時間で実現でき，BWT のチェックも $O(n)$ 時間で行えるので，極大部分文字列の列挙は線形時間で行うことができる．文字列中に含まれるすべての部分文字列の統計量は極大部分文字列の統計量として求めることができる．

極大部分文字列は，$2n-1$ 個よりも実際はかなり小さいことが多い．また，極大部分文字列を列挙する際に，頻度でフィルタリングを行うことができる．たとえば，10 回以上出現する部分文字列のみを列挙するといったことができる．

こうした極大部分文字列を介して，すべての部分文字列を特徴情報として利用し文書分類の学習や文書クラスタリングを効率的に行うことができる．詳しくは[27]を参照のこと．

8.2 文書集合の統計量

文書集合での部分文字列のさまざまな統計量を効率的に求める問題を考える．はじめに基本的な統計量を導入する．

定義 8.1（文書統計量）　文字列 Q と文書 d が与えられたとき，文書統計量を次のように定義する．

- $\mathbf{tf}(d, Q)$: Q が文書 d 中で出現している回数.
- $\mathbf{bin}(d, Q)$: I $(\mathbf{tf}(d, Q)>0)$, つまり 1 度でも出現していれば 1, そうでなければ 0.

□

例えば，$d=$"*banana*" の場合，$\mathbf{tf}(d,$ "a"$)=3$, $\mathbf{tf}(d,$ "na"$)=2$, $\mathbf{bin}(d,$ "na"$)=1$ である．

定義 8.2（文書集合統計量）　文書集合 $\{d_i\}$ と各文書の重み $\{w_i\}$, $w_i \in R$ が与えられているとき，文書集合統計量を次のように定義する．

- $\mathbf{df}(Q) = \sum_{d_i} \mathbf{bin}(d_i, Q)$: Q の出現文書数
- $\mathbf{mdf}(Q) = \sum_{d_i} \mathbf{tf}(d_i, Q)$: Q の重複有出現文書数
- $\mathbf{wmdf}(Q) = \sum_{d_i} w_i \mathbf{tf}(d_i, Q)$: Q の重み付き重複有出現文書数

□

こうした統計量はテキストマイニングや，文書分類，クラスタリングなどさまざまなタスクで利用できる．

はじめに，接尾辞配列を用いて先ほどの問題がどのように解けるのかを考えてみる．各文書をつなげた文字列 $T{=}d_1\$d_2\$\cdots$ を考え，T に対する接尾辞配列 SA を構築する．クエリ文字列 Q がヒットした接尾辞配列上の範囲が $SA[sp, ep)$ であったとする．このとき，重複有出現文書数は，T 上での出現回数 $\mathbf{mdf}(Q){=}ep{-}sp$ として求められる．

次に，$\mathbf{df}(Q)$ を求めることを考えてみよう．クエリ Q の T 中のヒット位置を求めた後，各ヒット位置が，どの文書に所属していたかは，T 中の文書開始位置を記録した長さ $|T|$ のビット配列を用意し，それに対する完備辞書を構築すれば定数時間で求めることができる．しかしヒット文書数がヒット位置数に比べて少ない場合は非効率である．ヒット文書数に比例する計算量で求められる方法を以降で考えていく．

文書配列 $D[0, |T|)$ を，$D[i]$ が $SA[i]$ の接尾辞が開始する文書番号を保持したような配列とする．次に配列 $P[0, |T|)$ を $P[i]$ を $D[i]$ と同じ文書で直前に出現した位置，つまり $P[i]{=}\max\{j{<}i \mid D[i]{=}D[j]\}$ であるような配列とする．もしそのような位置が見つからない場合は $P[i]{=}{-}1$ とする．配列 P は，同じ文書に対応する接尾辞の位置をつなげたリストであると考えられる．

このとき，ヒットしている各文書ごとに考えると，$[sp, ep)$ 中にある文書が複数回出現している場合は，$P[sp, ep)$ のうち，最初の 1 個だけが sp よりも小さい値を指しており，それ以外は，sp 以上の値を指しているはずである．よって $P[sp, ep)$ 中で sp 未満の値を持っている個数が重複がない出現文書数であり，それらの位置での文書番号がヒット文書番号集合である．このような $P[sp, ep)$ 中で，sp よりも小さい値が何回出現しているかは，P に対してウェーブレット木を構築すれば調べられる．この操作は $\mathbf{df}(Q){=}\mathbf{rangefreq}(P,$

$sp, ep, 0, sp)$ である．この操作は u を文書数だとしたとき，$O(\log u)$ 時間で求めることができる．この計算量はヒット位置数，ヒット文書によらないことに注意する．

さて，D のウェーブレット木の作業領域量は $n \lg n$ ビットであるが，各ビット列を RRR による表現で格納すれば $n \lg u$ ビットで格納できることが示されている．

では最後に，重み付き重複有出現文書数を考えてみる．配列 $S[0, n)$ を用意し，$S[i]=\sum_{j=0}^{i-1} w_{D[j]}$ とする．このとき，$S[ep]-S[sp]$ は，

$$\begin{aligned} S[ep]-S[sp] &= \sum_{j=sp}^{ep-1} w_{D[i]} \\ &= \sum_{d_i} w_i \mathbf{tf}(D_i, Q) \\ &= \mathbf{wmdf}(Q) \end{aligned} \tag{8.4}$$

となる．以上をまとめると，文書集合統計量は

$$\mathbf{df}(Q) = \mathbf{rangefreq}(P, sp, ep, 0, sp) \tag{8.5}$$

$$\mathbf{mdf}(Q) = ep-sp \tag{8.6}$$

$$\mathbf{wmdf}(Q) = S[ep]-S[sp] \tag{8.7}$$

として求められる．この他にも，極大部分文字列や接尾辞木を利用して，さまざまな文字列操作が効率よく処理できることが知られている[13]．

8.3　文書配列を利用した統計量の計算

次に，文書配列を利用した統計量の計算についてみてみる．各文書をつなげた文字列 $T=d_1\$d_2\$\cdots$ を考え，T に対する接尾辞配列 SA を構築する，また，$n=|T|$ とする．前章と同じように文書配列 $D[0, n)$ を $D[i]$ は接尾辞 $SA[i]$ が開始する文書番号を保持するようにする．この文書配列 D に対してウェーブレット木を構築する．

クエリ文字列 Q のヒット範囲が $SA[sp, ep)$ であったとする．これは FM-index などの全文索引を利用して $O(m \log \sigma)$ 時間で求められる．このとき，

文書配列 $D[sp, ep)$ がヒットした文書番号である．文書中の頻度 $\mathbf{tf}(D, Q)$ が大きい順に文書を列挙したい場合は，$D[sp, ep)$ 中の出現数が多い文書順に列挙すればよい．これはウェーブレット木を利用すれば，$\mathbf{topk}(D, sp, ep, k)$ として求められ，一緒に $\mathbf{tf}(d, Q)$ も求められる．計算量は $D[sp, ep)$ 中の異なる文書数を h としたとき，最悪 $\mathrm{O}(h \log \sigma)$ 時間必要だが，多くの場合 $\mathrm{O}(k \log \sigma)$ 時間で求めることができる．

次に，文書に優先度がついており，ヒットした中で優先度の高い順に列挙したい場合を考えてみよう．例えば，最新文書の順に列挙したい場合は新しい文書であるほど文書番号が大きくなるようにし，文書のスコアが大きい順に列挙したい場合は，スコアが小さい順に文書番号をつければよい．

このようにすれば後は $D[sp, ep)$ 中で文書番号が大きい順にヒットした文書を列挙すればよい．これは，ウェーブレット木を利用すれば $\mathbf{rangemaxk}(D, sp, ep)$ で求めることができ，計算量は $\mathrm{O}(k \log u)$ 時間である．

また，複数のクエリ Q_1, Q_2 が与えられたとき，これらが共通して出現する文書を列挙できる．Q_1, Q_2 に対応する文書配列の範囲が $D[sp_1, ep_1)$, $D[sp_2, ep_2)$ のとき，$\mathbf{intersect}(D, sp_1, ep_1, sp_2, ep_2)$ として共通して出現する文書を列挙できる．

この文書番号を格納するには $n \lg u$ ビット必要であるが圧縮可能であり，元テキストとほとんど変わらない容量で格納可能である．この他，ウェーブレット木を利用することでさまざまな文書集合上の操作が可能である[10]．

参考文献

[1] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management (IPM)*, pp.392–404, 2012.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. *Technical Report* 124, Digital Equipment Corporation, 1994.

[3] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, pp.176–187, 2008.

[4] F. Claude and G. Navarro. The wavelet matrix. In *Proceedings of the 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, pp.167–179, 2012.

[5] F. Claude, P. K. Nicholson, and D. Seco. Space efficient wavelet tree construction. In *Proceedings of the 18th international conference on String processing and information retrieval (SPIRE)*, pp.185–196, 2011.

[6] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:12, pp.1–19, 2009.

[7] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp.184–196, 2005.

[8] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proceedings of the 18th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pp.690–696, 2007.

[9] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):10, pp.535–542, 2010.

[10] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comput. Sci.*, 426-427, pp.25–41, 2012.

[11] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp.295–306, 2006.

[12] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algo-*

rithms (SODA), pp.841–850, 2003.

[13] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[14] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of burrows wheeler-based compression. *Theor. Comput. Sci.*, 387(3), pp.220–235, 2007.

[15] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in fm-indexes. In *Proceedings of the 18th international conference on String processing and information retrieval (SPIRE)*, pp.174–184, 2011.

[16] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of CPM*, pp.181–192, 2001.

[17] S. Rao Kosaraju and G. Manzini. Compression of low entropy strings with lempel–ziv algorithms. *SIAM J. Comput.*, 29(3), pp.893–911, 1999.

[18] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *Proceedings of the 20th Data Compression Conference (DCC)*, pp.239–248, 2010.

[19] P. D. Michailidis and K. G. Margaritis. On-line string matching algorithms: survey and experimental results. *International Journal of Computer Mathmatics*, 76(4), pp.411–434, 2001.

[20] A. Moffat and A. Turpin. *Compression and Coding Algorithms.* The Kluwer international series in engineering and computer science. Kluwer Academic, 2002.

[21] V. Markin and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. *Technical report.* University of Helsinki, 2004.

[22] G. Navarro. Wavelet trees for all. In *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp.2–26, 2012.

[23] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39, 2007.

[24] G. Navarro and L. M. S. Russo. Space-efficient data-analysis queries on grids. In *Proceedings of the 22nd international conference on Algorithms and Computation (ISAAC)*, pp.323–332, 2011.

[25] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Comput.*, 60(10), pp.1471–1484, 2011.

[26] D. Okanohara and K. Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pp.90–101, 2009.

[27] D. Okanohara and J. Tsujii. Text categorization with all substring features. In *Proceedings of the SIAM International Conference on Data Mining*

(SDM), pp.838–846, 2009.

[28] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp.233–242, 2002.

[29] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms (TALG)*, 7:53, pp.1–34, 2011.

[30] D. Shkarin. PPM: One step to practicality. In *Proceedings of the Data Compression Conference (DCC)*, pp.202–211, 2002.

[31] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web (WWW)*, pp.387–396, 2008.

索　引

ナ　行

ハ　行

マ　行

ラ　行

岡野原大輔

1982 年生まれ．2010 年 3 月，東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程を修了，情報理工学博士．2006 年 3 月に(株)Preferred Infrastructure を共同で創業，現在，同社取締役副社長．主な受賞歴は，IPA 未踏ソフトウェア創造事業スーパークリエータ認定(2005 年)，東京大学総長賞(2007 年)，言語処理学会年次大会優秀発表賞(2009 年，2010 年)など．統計的自然言語処理，機械学習，簡潔データ構造，大規模データ解析に興味を持つ．

シリーズ　確率と情報の科学

高速文字列解析の世界――データ圧縮・全文検索・テキストマイニング

2012 年 12 月 26 日　第 1 刷発行
2013 年　2 月 25 日　第 3 刷発行

著　者　岡野原大輔（おかのはらだいすけ）

発行者　山口昭男

発行所　〒101-8002　東京都千代田区一ツ橋 2-5-5　株式会社 岩波書店　電話案内 03-5210-4000
http://www.iwanami.co.jp/

印刷・法令印刷　カバー・半七印刷　製本・松岳社

　ISBN 978-4-00-006974-8